Timed Circuit Verification Using TEL Structures

Wendy Belluomini, *Member, IEEE*, Chris J. Myers, *Member, IEEE*, and H. Peter Hofstee, *Member, IEEE*

***Abstract*—Recent design examples have shown that significant performance gains are realized when circuit designers are allowed to make aggressive timing assumptions. Circuit correctness in these aggressive styles is highly timing dependent and, in industry, they are typically designed by hand. In order to automate the process of designing and verifying timed circuits, algorithms for their synthesis and verification are necessary. This paper presents timed event/level (TEL) structures, a specification formalism for timed circuits that corresponds directly to gate-level circuits. It also presents an algorithm based on partially ordered sets to make the state-space exploration of TEL structures more tractable. The combination of the new specification method and algorithm significantly improves efficiency for gate-level timing verification. Results on a number of circuits, including many from the recently published gigahertz unit Test Site (guTS) processor from IBM indicate that modules of significant size can be verified using a level of abstraction that preserves the interesting timing properties of the circuit. Accurate circuit level verification allows the designer to include less margin in the design, which can lead to increased performance.**

I. Introduction

In ORDER to increase performance, circuit designers are beginning to move away from traditional synchronous designs based on static logic. Recent designs, such as the Intel RAPPID instruction length decoder [1] and the IBM guTS microprocessor [2], have shown that large performance gains can be realized using aggressive circuit styles, which make many timing assumptions. The RAPPID chip is an asynchronous implementation of an instruction length decoder for a Pentium II instruction set. It achieves a three times performance improvement while dissipating half the power of the synchronous implementation on the same process. The guTS microprocessor is a synchronous implementation of a PowerPC instruction set running at 1 GHz on a 0.25-μm CMOS process available in 1997. Although both designs achieve significant performance gains, they are experimental designs. Many obstacles need to be overcome before the circuit styles developed in these designs can be used in production. One of the main obstacles is the lack of design automation for timed design styles.

Although the Intel design is asynchronous and the IBM design is synchronous, the timing analysis problems they create for synthesis and verification tools are similar. The circuits used in the guTS processor are synchronous, but their local behavior is asynchronous, and the timing constraints that are required for them to work correctly are quite complex. Existing synchronous static timing analysis methods can be adapted to analyze this type of circuit [3]–[5], but they have some limitations. The approach presented in [3] extends the static timing methodology by changing the standard two-event per signal model to a four-event per signal model. This allows all of the relevant timing constraints on a domino gate to be verified. However, since the method considers only topological delay and not Boolean behavior, it can be overly pessimistic. The method presented in [5] is successful in verifying a large high performance chip. It adds some Boolean behavior to the topological delay calculations in order to improve accuracy, but may still be conservative. The technique presented in [4] is designed to verify self-resetting or delayed-reset circuits at the macro level. A designer must determine an interface specification for each macro through simulation. The timing analysis tool then determines if the combination of all the macros correctly implements all of the interfaces. This works well for chip level timing verification but the interfaces specified by the designer are never formally verified. A tool that correctly verifies that the gates inside the macro work within the specified interface is required to complete the verification and this appears to be the ideal place to use a tool designed for asynchronous circuits.

Since this timing verification problem deals primarily with gate-level circuits, a specification method that corresponds directly to logic gates is needed. There are currently two general approaches to specifying the behavior of time dependent circuits, time (or timed) Petri nets [6], and timed automata [7]. Both of these approaches were first proposed to model concurrent software systems and have since been applied to the synthesis and verification of time dependent asynchronous circuits [8]–[12]. They both have drawbacks when applied to circuits. The Petri net model lacks support for Boolean conditions, which are central to the specification of gate-level circuits since gates respond to signal levels and not signal transitions. Although the Petri net model is expressive enough to specify level-based behavior, it requires adding an extensive number of feedback arcs and generally increases the complexity of the Petri net. This increased complexity results in increased synthesis and verification time. Additionally, the method in [11] produces time separations between events instead of directly calculating the state space. Although these separations can be used to derive the state space, it may not be exact since there may be time separations between events that are state dependent. The other approach, based on timed automata, is in fact more expressive than timed event/level (TEL) structures. However, TEL structures provide more powerful primitives to encode states, implicitly leading to more succinct representations. Also, the generality of timed automata complicates the timed state-space exploration procedure. By restricting our class of specifications to those which can be

Manuscript received September 10, 1999; revised March 20, 2000. This work was supported by the National Science Foundation CAREER award MIP-9625014 and a Traineeship award, by the SRC under Contract 97-DJ-487, and by the DARPA ASSERT Fellowship. This paper was recommended by Associate Editor D. Dill.

W. Belluomini and H. P. Hofstee are with the IBM Austin Research Laboratory, Austin, TX 78758 USA.

C. J. Myers is with the Department of Electrical Engineering, University of Utah, Salt Lake City, UT 84112 USA.

Publisher Item Identifier S 0278-0070(01)00358-X.

represented using TEL structures, we have been able to develop a more efficient analysis method.

This paper introduces a new specification method, TEL *structures,* which are created specifically to represent circuits. TEL structures have been shown to be easy to produce from a higher level language, such as VHDL [13], and are useful in representing gate-level circuits [14]. We also present an adaptation of the POSET algorithm, first presented for Petri nets in [15], to the analysis of TEL structures. The combination of the efficient representation for circuits provided by TEL structures and the state-space reduction resulting from the POSET algorithm results in an order of magnitude increase in speed and memory performance over previously published timed state-space exploration algorithms. The improvement makes it possible to analyze circuits of considerable complexity as illustrated by verification of timed asynchronous circuits in the Stari communication protocol and the Intel instruction length decoder, and by the verification of the timed synchronous circuits from the IBM guTS processor.

II. TEL Structures

Timed Event Level (TEL) structures are designed with two goals. The first is to correspond as directly as possible to gate-level circuit behavior. Circuit specifications typically use signal transitions to specify sequencing and signal levels to specify data, so TEL structures allow both events and levels to be specified. The second goal is to provide a specification formalism that can be generated automatically from a higher level language. The flexibility of TEL structures makes their automatic construction from a standard hardware description language like VHDL possible [13]. Compiling VHDL to a Petri net is difficult since VHDL specifications contain both level and event-based behavior.

TEL structures are based on timed Event Rule (ER) structures [16], which are fundamentally acyclic. Cyclic specifications are represented by infinite timed ER structures, and state-space exploration is done by dynamically creating the unrolling of the specification until no new Boolean states are possible. This type of acyclic semantics can also be used for TEL structures [17], but in order to make them more similar to the widely accepted specification methods such as Petri nets, TEL structures are defined here as cyclic structures.

A TEL structure is a 6-tuple $T = \langle N, s_0, A, E, R, \# \rangle$ where

1) *N is the set of signals;*
2) *$s_0 = \{0, 1\}^{|N|}$ is the initial state;*
3) *$A \subseteq N \times \{+, -\} \cup \$$ is the set of actions;*
4) *$E \subseteq A \times (\mathcal{N} = \{0, 1, 2, \ldots\})$ is the set of events;*
5) *$R \subseteq E \times E \times \mathcal{N} \times (\mathcal{N} \cup \{\infty\}) \times (b\colon \{0, 1\}^{|N|} \rightarrow \{0, 1\})$ is the set of rules;*
6) *$C \subseteq E \times E \times \mathcal{N} \times (\mathcal{N} \cup \{\infty\}) \times (b\colon \{0, 1\}^{|N|} \rightarrow \{0, 1\})$ is the set of constraint rules;*
7) *R_0 is the set of initially marked rules;*
8) *$\# \subseteq E \times E$ is the conflict relation.*

The signal set N contains the wires in the circuit specification. The state s_0 contains the initial value of each signal in N. The action set A contains for each signal x in N a rising transition $x+$ and a falling transition $x-$ along with the sequencing event \$, which is used to indicate an action that does not cause a signal transition. The event set E contains actions paired with instance indices (i.e., $\langle a, i \rangle$), which are used to distinguish multiple instances of a given signal transition within the specification. For example, there may be two possible situations in which a signal x can rise in a specification. These rising actions on x are distinguished by having two events, $\langle x+, 1 \rangle$ and $\langle x+, 2 \rangle$. Pairing actions with instance indices allows an arbitrary number of events to be created from each action, including the sequencing action \$. Sequencing events are often used to express nondeterminism where a signal may or may not transition. Although formally the definition requires that all sequencing events be of the form $\langle \$, i \rangle$, where i is an integer, sequencing events of the form \$s, where s is a string, are used in this paper in order to make the purpose of the sequencing event more clear.

Rules represent causality between events. Each rule r is of the form $\langle e, f, l, u, b \rangle$ where

- e enabling event;
- f enabled event;
- $\langle l, u \rangle$ bounded timing constraint;
- b Boolean function over the signals in N.

A rule becomes marked when its enabling event fires. It is *enabled* if its enabling event has occurred and its Boolean function is true in the current state. There are two possible semantics concerning the enabling of a rule. In one semantics, referred to as *nondisabling semantics*, once a rule becomes enabled, it cannot lose its enabling due to a change in the state. In the other semantics, referred to as *disabling semantics*, a rule can become enabled and then lose its enabling. This can occur when another event fires, resulting in a state where the Boolean function is no longer true. A single specification can include rules with both types of semantics. Nondisabling semantics are typically used to specify environment behavior and disabling semantics are typically used to specify logic gates. For the purposes of verification, the disabling of a Boolean expression on a disabling rule is assumed to correspond to a failure since it corresponds to a glitch on the input to a gate. A rule is *satisfied* if it has been at least l time units since it was enabled, and *expired* if it has been at least u time units since it was enabled. When a rule fires, it becomes unmarked. Excluding conflicts, an event cannot occur until every rule enabling it is satisfied, and it must occur before every rule enabling it has expired. This timing semantics, often referred to as "Type 2" timing semantics [18], means that the upper bounds on some rules may be exceeded. This timing semantics closely models gate-level circuits, where gate outputs transition some amount of time after their inputs change. It also eliminates the problem of maintaining proper causality when no upper bound can be exceeded, which is described in [19]. Constraint rules have the same structure as standard rules, but they are used to represent *requirements* on the relationships between events instead of causality between events. When an event fires, all of the constraint rules that enable it must be satisfied. If they are not, the algorithm generates a verification failure.

The conflict relation $\#$ is used to model disjunctive behavior and choice. When two events e and e' are in conflict (denoted

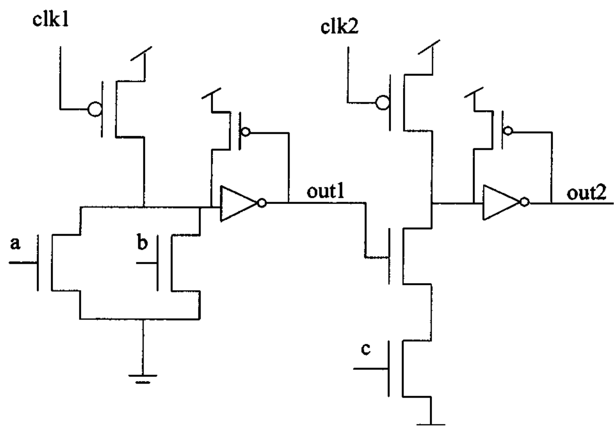

Fig. 1. Delayed-reset domino gate.

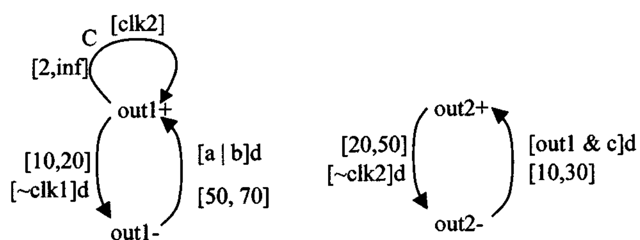

Fig. 2. TEL structure for the gate in Fig. 1.

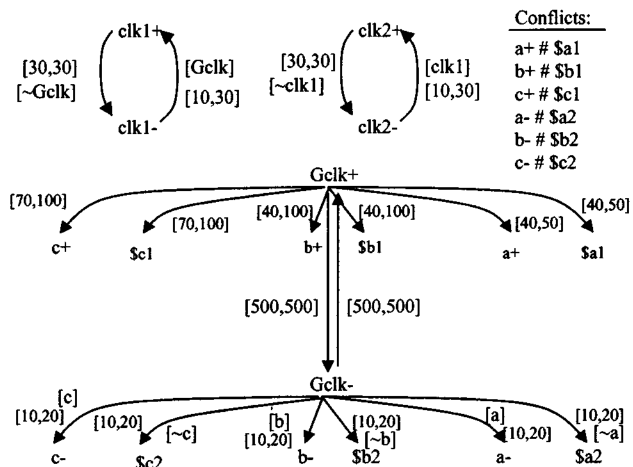

Fig. 3. Environment for domino gate.

$e \# e'$), this specifies that either e can occur or e' can occur, but not both. Taking the conflict relation into account, if two rules have the same enabled event and conflicting enabling events, then only one of the two mutually exclusive enabling events needs to occur to cause the enabled event. In the general case, an event is enabled when a maximal nonconflicting set of its enabling events has fired. The ability for an event to fire when only a subset of its enabling events have fired models a form of disjunctive causality. Events that are enabled by multiple conflicting events are similar to merge places in Petri nets. Choice is modeled when two rules have the same enabling event and conflicting enabled events. In this case, only one of the enabled events can occur. An event e, which is the enabling event of multiple rules that have conflicting enabled events, is similar to a choice place in a Petri net. Every pairwise conflict in the TEL structure must be specified, but this does not cause a problem for the user since TEL structures are typically generated from a higher level input language, such as VHDL [13].

A. Examples

Fig. 1 shows an example of a delayed-reset domino gate. This type of circuit is used extensively in the guTS processor to gain higher performance. The gate computes the function $(a \vee b) \wedge c$ in two stages. The first stage computes $a \vee b$ while $clk1$ is high, and the next stage computes $out1 \wedge c$ while $clk2$ is high. Both gates precharge while their respective clocks are low. Since neither n-stack has a "foot" transistor to ensure that the path to ground is turned off during the precharge phase, the timing of the circuit must guarantee that all the inputs to the gate are low by the time the local clock for each stage falls.

The TEL structure representation for the domino gate is shown in Fig. 2. It includes one rising and one falling event for each signal. The specification for the gate corresponds directly to the structure of the circuit. The signal $out1$ must rise between 50 and 70 time units after the Boolean expression $a \vee b$ becomes true, and it falls ten to 20 time units after $clk1$ falls. The signal $out2$ rises between ten and 30 time units after $out1 \wedge c$ becomes true and it falls 20 to 50 time units after $clk2$ falls. The TEL structures for the gate outputs contain disabling rules, indicated by a d next to the Boolean expression. Once these expressions are true, any event firing that makes them false causes a failure. The TEL structure also contains a constraint rule, marked with a "C." The constraint rule requires that $clk2$ has been high for at least two time units when $out1$ rises. This ensures that the pulldown stack for the second stage of the gate is turned off before pulldown stack can be turned on.

In order to verify the circuit, the analysis tool needs to know what inputs it may receive. This is specified by providing TEL structures for the environment of the circuit. The environment TEL structures for the domino gate are shown in Fig. 3. The specification indicates that there is a global clock $Gclk$, which rises 500 time units after it falls and falls 500 time units after it rises. The $Gclk$ signal controls the firing time of the two local clocks, $clk1$ and $clk2$. The rule $[clk1-, clk1+]$ becomes enabled when $Gclk$ is true and it becomes satisfied after ten time units have passed. It becomes expired after 30 time units and thus must fire between ten and 30 time units after $Gclk$ rises. The other local clock $clk2$ is similar. The inputs to the gate a, b, and c nondeterministically rise some time after the clock rises. The nondeterminism is modeled using the conflict relation and sequencing events. Each rising event on an input conflicts with a corresponding sequencing event. Since the rising event and the sequencing event conflict, only one of them can occur. If the rising event for a signal fires, the signal rises in that clock cycle; if the sequencing event fires, it does not. A falling transition on the global clock is followed by falling transitions on all of the inputs that have risen. Boolean guards are used to determine if a signal has risen in the most recent cycle. Sequencing events and conflicts are again used to deal with the nondeterminism. If an input signal rises on the rising edge of $Gclk$, then a falling event for that signal must occur when $Gclk$ falls. Otherwise, a conflicting sequencing

event fires, preventing the falling event on the input signal from becoming enabled as soon as that signal rises again. This environment allows all possible values of the input signals to be applied to the circuit nondeterministically. If certain input patterns are known to be unreachable, then the circuit can be verified in a more restricted environment. Although the TEL structure is readable for a small circuit, it would be difficult to specify a large macro at this level. Our synthesis and verification tool, ATACS, provides support for two higher level input languages, VHDL and CSP. Designers can specify circuits in these languages, and they are compiled into TEL structures using techniques described in [13] and [16].

B. Timed Firing Sequences

The behavior specified by a TEL structure is defined with three types of operations: firing of rules, firing of events, and advancement of time. A time valued clock c_i is associated with each enabled rule r_i. A rule can fire when the clock meets the lower bound on the rule, and must fire when the clock reaches the upper bound on the rule. Using these semantics, the age of a clock never exceeds the upper bound of its associated rule. The firing of a rule may not immediately result in the firing of an event. An event fires when a *sufficient set* of the rules that enable it have fired. If all of the rules enabling an event e have nonconflicting enabling events, then e's sufficient set is all of the rules that enable it. If some of the rules enabling e have conflicting enabling events, then e has a number of different sufficient sets. For a set of rules R_s to be sufficient to fire e, all rules that enable e and are not in R_s must have enabling events that conflict with the enabling event of some rule in R_s. Events fire simultaneously with the last rule firing, which creates a sufficient set of fired rules. Time is advanced using a function *max_advance*, which returns the maximum amount of time that can pass before a rule must fire or exceed its upper bound. These semantics define a set of firing sequences that contain both rule and event firings, where event firings are placed in the sequence immediately following the firing of its final enabling rule. In order for the analysis algorithm presented here to succeed in finding the state space of a TEL structure, it must be *one-safe*. In a one-safe TEL structure, when the enabling event of a rule fires, it cannot fire again until either the enabled event of the rule fires or an event that conflicts with its enabled event fires. This property is similar to the one-safe property on Petri nets, which prevents places from containing multiple tokens.

The set of behaviors of a TEL structure is defined by a set of sequences $\Sigma \in ((R^*)(E^*))^*$, where each firing (rule or event) is numbered sequentially. In order to simplify the notation, shorthand operations for dealing with firing sequences need to be defined. The function L is used to map an instance of a rule or event in the firing sequence back to the corresponding rule or event in the original specification. The $\in$ operator is used to indicate whether a given rule or event firing occurs in a sequence. For example $x \in \sigma$ indicates that firing x occurs in sequence σ. Also, the functions l and u are used to return the lower and upper bound on a rule. Finally, it is useful to define a *choice_set* for each rule $r = \langle e, f, l, u, b\rangle$. The choice set of r contains all events, which are enabled by e and conflict with f.

Definition II.1: The choice set of a rule $r = \langle e, f, l, u, b\rangle$ is defined as follows:

$$\text{choice_set}(r) = \{f' \in E \mid \exists r' = \langle e, f', l', u', b'\rangle \in R \wedge f' \# f\}.$$

When the event f fires, all of the events in the choice set of r require another firing of e before they have a chance to fire. Events that are not in the choice set of r do not require another firing of e in order to fire. When an event in the choice set of a marked rule fires, the rule becomes unmarked.

The state space of a TEL structure is found by exploring firing sequences of events and rules. The Boolean state, which is used to evaluate the Boolean expressions associated with rules, is defined by the rule firing sequence being explored σ. The state resulting from a rule firing sequence $\phi(\sigma)$ is simply the state that results when the firing sequence is executed starting from the initial state s_0. We can now formally define what it means for a rule r to be enabled by a firing sequence σ.

Definition II.2: A rule $r = \langle e, f, l, u, b\rangle \in enabled(\sigma_{0\cdots n})$ if one of the following conditions is true:

1) $(r \in R_0) \wedge (\neg\exists\sigma_j \in \sigma_{0\cdots n}\colon L(\sigma_j) = r) \wedge$
$(\neg\exists\sigma_j \in \sigma_{0\cdots n}\colon L(\sigma_j) \in choice_set(r)) \wedge$
$(b(\phi(\sigma_{0\cdots n})) \vee$
$(\text{nondisabling}(r) \wedge \exists\sigma_j \in \sigma_{0\cdots n}\colon b(\phi(\sigma_{0\cdots j})))$

2) $\exists\sigma_i \in \sigma_{0\cdots n}\colon ((L(\sigma_i) = e) \wedge$
$(\neg\exists\sigma_j \in \sigma_{i+1\cdots n}\colon L(\sigma_j) = r) \wedge$
$(\neg\exists\sigma_k \in \sigma_{i+1\cdots n}\colon L(\sigma_k) \in \text{choice_set}(r)) \wedge$
$(b(\phi(\sigma_{0\cdots n})) \vee \text{nondisabling}(r) \wedge$
$\exists\sigma_l \in \sigma_{i+1\cdots n}\colon b(\phi(\sigma_{0\cdots l})))))$

The first condition in the definition deals with rules that are initially marked. In order to satisfy the first condition, a rule must be initially marked (i.e., $r \in R_0$) and there must not be any other firing of the rule in the firing sequence (i.e., $\neg\exists\sigma_j \in \sigma_{0\cdots n}\colon L(\sigma_j) = r$). There also must not be any other event firings in the sequence that would cause this rule to lose its chance to fire due to conflict [i.e., $\neg\exists\sigma_j \in \sigma_{0\cdots n}\colon L(\sigma_j) \in choice_set(r)$]. Finally, the Boolean expression on the rule must either be satisfied by the current firing sequence or be satisfied at some point in the current firing sequence for a nondisabling rule [i.e., $(b(\phi(\sigma_{0\cdots n})) \vee (\text{nondisabling}(r) \wedge \exists\sigma_j \in \sigma\colon b(\phi(\sigma_{0\cdots j})))$]. This distinction is made since nondisabling rules only require that the Boolean expression become true at some point before the rule fires. The second condition deals with all rule enablings other than the first firings of initially marked rules. In order for the second condition to hold, the firing sequence must contain a firing of the enabling event of the rule [i.e., $\exists\sigma_i \in \sigma_{0\cdots n}\colon L(\sigma_i) = e$] and it must not contain a firing of the rule that occurs after the firing of the enabling event [i.e., $\neg\exists\sigma_j \in \sigma_{i+1\cdots n}\colon L(\sigma_j) = r$]. The firing sequence also must not contain a firing of an event in the choice set of r that occurs after the firing of e [i.e., $\neg\exists\sigma_k \in \sigma_{i+1\cdots n}\colon L(\sigma_k) \in \text{choice_set}(r)$]. Finally the Boolean expression on the rule must either be satisfied by the current firing sequence or, if the rule is nondisabling, it must have been satisfied at some point in the sequence after the firing of the enabling event [i.e., $b(\phi(\sigma_{0\cdots n})) \vee (\text{nondisabling}(r) \wedge \exists\sigma_l \in \sigma_{l+1\cdots n}\colon b(\phi(\sigma_{0\cdots l})))$].

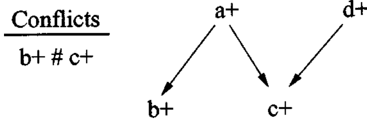

Fig. 4. Conflict behavior.

When a sufficient set of rules has fired in the sequence, an event becomes enabled to fire. When an event fires, it "uses" the rule firings. Therefore, we need to define when a rule firing can be used to fire an event.

Definition II.3: The *usable* relation on σ_i: $L(\sigma_i) = \langle e, f, l, u, b\rangle$ and $\sigma_{0\cdots n}$ is defined as follows:

$$usable(\sigma_i, \sigma_{0\cdots n}) \Leftrightarrow \neg\exists\sigma_j \in \sigma_{i+1\cdots n}: (L(\sigma_j) = f) \vee (L(\sigma_j) \in \text{choice_set}(L(\sigma_i))).$$

This definition means that a rule firing is usable until its enabled event fires or an event in its choice set fires. A rule $r = \langle e, f, l, u, b\rangle$ remains usable when an event f' that conflicts with f fires, if f' and f do not share e as an enabling event. For example, consider the TEL structure in Fig. 4, and assume that $a+$ and $d+$ have fired. A firing of $a+ \rightarrow c+$ is made unusable by the firing of event $b+$ since $b+$ is in the choice set of $a+ \rightarrow c+$. However, the firing of $b+$ does not make a firing of $d+ \rightarrow c+$ unusable. This distinction is made since another firing of $a+$ is necessary before $c+$ can fire, but another firing of $d+$ is not necessary before $c+$ can fire.

Events fire when there is a sufficient set of usable rules.

Definition II.4: The set of fireable events of a firing sequence $\sigma_{0\cdots n}$ is defined as follows:

$$fireable(\sigma) = \{f \in E | \forall r = \langle e, f, l, u, b\rangle \in R: \exists\sigma_i \in \sigma_{0\cdots n}: L(\sigma_i) = r \wedge usable(\sigma_i, \sigma_{0\cdots n}) \vee \exists\sigma_j \in \sigma_{0\cdots n}: L(\sigma_j) = \langle e', f, l, u, b\rangle \wedge e' \# e \wedge usable(\sigma_j, \sigma_{0\cdots n})\}.$$

The fireable set contains all events which have a sufficient set of usable rules in the firing sequence. All of the rules that enable an event must either have a usable firing in σ or have an enabling event which conflicts with a rule that has a usable firing in σ.

Definition II.4 allows us to define the set of sequences that are allowed by the TEL structure $\Sigma \in ((R^*)(E^*))^*$ as follows:

Definition II.5: A sequence $\sigma \in \Sigma$ if and only if $\forall\sigma_i$

1) $L(\sigma_i) \in R \Rightarrow L(\sigma_i) \in enabled(\sigma_{0\cdots i-1})$;
2) $L(\sigma_i) \in E \Rightarrow L(\sigma_i) \in \text{fireable}(\sigma_{0\cdots i-1})$;
3) $L(\sigma_i) \in R \wedge \text{fireable}(\sigma_{0\cdots i}) \neq \emptyset \Rightarrow \sigma_{i+1} \in \text{fireable}(\sigma_{0\cdots i})$.

The first requirement states that rules must be enabled when they fire. The second requirement of this definition states that all events must be in the fireable set when they fire. The third requirement is that if the fireable set of a rule firing is not empty, an event in the fireable set must follow it in the sequence.

Each rule firing σ_i can be associated with the event firing that enabled the rule by the causal event function E_c defined as follows:

Definition II.6: $E_c(\sigma_i, \sigma) = \sigma_j$ where j is the maximum value less than i for which

$$L(\sigma_i) \notin enabled(\sigma_{0\cdots j-1}) \quad \text{and} \quad L(\sigma_i) \in enabled(\sigma_{0\cdots j}).$$

This means that the causal event for a rule firing is the event firing that causes the rule to become enabled. This event may either be the enabling event for the rule or it may be an event that changes the value of a signal, which causes the Boolean expression associated with the rule to evaluate to true.

Any sequence can be given a *timing assignment* τ, which maps an event to the time at which it occurs. For each sequence $\sigma \in \Sigma$, there is a set of *valid* timing assignments, referred to as $valid(\sigma)$.

Definition II.7: A timing assignment τ is valid for a sequence σ if

$\forall\sigma_i \in \sigma$:

$$\tau(\sigma_i) \leq \tau(\sigma_{i+1}) \wedge L(\sigma_i) \in E \Rightarrow \tau(\sigma_i) = \tau(\sigma_{i-1}) \wedge L(\sigma_i) \in R \Rightarrow \tau(E_c(\sigma_i, \sigma)) + l(L(\sigma_i)) \leq \tau(\sigma_i) \leq \tau(E_c(\sigma_i, \sigma)) + u(L(\sigma_i)).$$

This means that a timing assignment is valid if it corresponds to the order of the firing sequence, all events fire simultaneously with the rule immediately preceding their firing, and rules fire between their lower and upper bounds after their causal event. A firing sequence $\sigma \in \Sigma$ is reachable in the specification TEL structure if and only if it can be given a valid timing assignment.

III. POSET ALGORITHM

In order to determine if the set of firing sequences allowed by a TEL structure results in a failure, it is necessary to find the timed state-space of the specification. The difficulty in doing this is controlling the state explosion problem. A number of techniques have been proposed to deal with state explosion. One approach is to minimize the number of interleavings due to concurrency that are explored. These techniques include stubborn sets [20], partial orders [21], or unfoldings [22]. While they have been successful, they only deal with untimed systems.

The size of the timed state space is highly dependent on the time representation that is used. Timing behavior can either be modeled continuously (i.e., dense-time), where the timers in the system can take on any value between their lower and upper bounds, or discretely, where timers can only take on values that are multiples of a discretization constant. Discrete time has the advantage that the timing analysis technique is simpler and implicit techniques can be easily applied to improve performance as shown in [12] and [23]. However, the state space explodes if the delay ranges are large and the discretization constant is set small enough to ensure exact exploration of the state space. For example, a delay range of 117 to 269 has 153 discrete states if the discretization constant is set to one. Although the discretization constant can be larger than one if there is a larger number that divides all of the numbers used for delay ranges, this does not happen very often when delay numbers from actual circuit data are used.

Continuous time techniques eliminate the need for a discretization constant by breaking the infinite continuous timed state space into equivalence classes. All timing assignments within an equivalence class lead to the same behavior and do not need to be explored separately. In the *unit-cube* (or region) approach [7], timed states with the same integral clock values

and a particular linear ordering of the fractional values of the clocks are considered equivalent. Although this approach eliminates the need to discretize time, the number of timed states is dependent on the size of the delay ranges and the number of concurrently enabled clocks. This state space can quickly explode for even relatively small systems.

Another approach to continuous time is to represent the equivalence classes as convex *geometric regions* (or zones) [24]–[26]. These geometric regions can be represented by sets of linear inequalities (also known as *difference bound matrices* [DBMs]). These larger equivalence classes can often result in smaller state spaces than those generated by the unit-cube approach. While geometric methods are efficient for some problems, their complexity can be worse than either discrete or unit-cube methods when analyzing highly concurrent systems. The number of geometric regions can explode with these approaches since each untimed state has at least one geometric region associated with it for every firing sequence that can result in that state. In highly concurrent systems, where many interleavings are possible, the number of geometric regions per untimed state can be huge. Some researchers [8]–[10], [27] have attacked this problem by reducing the number of interleavings explored using the partial order techniques developed for untimed systems. These algorithms compute a set of event firings that must be interleaved to ensure that the desired property is checked. Any event firings not in the set are not interleaved. This reduces the state space significantly for highly concurrent specifications. While reducing the number of interleavings is useful, in [8] and [9] one region is still required for every firing sequence explored to reach a state. If most interleavings need to be explored, these techniques could still result in state explosion. The algorithms from [10] and [27] do address the problem of generating a unique region for every firing sequence. However, since these techniques do not find the entire state space, they cannot be applied to synthesis. In order for existing algorithms to perform correct logic synthesis of timed asynchronous circuits, the entire reachable state space must be found [28]. If the synthesis algorithm is given an incomplete state space, it cannot be guaranteed to generate logic that correctly responds to all inputs to the circuit.

`Orbits`, presented in [29]–[31], takes a somewhat different approach. It reduces the number of regions per untimed state by using *partially ordered sets* (POSETs) of events rather than linear sequences to construct the geometric regions. The algorithm generates only one geometric region for any set of firing sequences that differ only in the firing order of concurrent events. This algorithm, shown in [30], results in very few geometric regions per untimed state. This algorithm differs from the partial order approaches in that it still finds a complete state space and improvement achieved by `Orbits` and is not dependent on the verification property. However, it is limited to specifications where the firing time of an event can only be controlled by a single predecessor event [known as the *single behavioral place (or rule) restriction*]. In some cases, the single behavioral rule restriction can be worked around through transformations on the initial graphs [16]; however, the transformations cause a large increase in the complexity of the graphs which need to be analyzed.

In [15] and [32], we present a new version of the POSET algorithm that applies to specifications without the single behavioral place restriction and in [15] we present its theoretical foundation using a timed Petri net model. The problem of analyzing a timed Petri net is isomorphic to the analysis of a TEL structure where all Boolean expressions **true**. The Petri net model is used in [15] because it is better known than the TEL structure model. Since the theory from [14] and [15] can be applied directly to the analysis of TEL structures without level expressions, this section assumes knowledge of that theory and concentrates on the complexities added to the theory by the Boolean expressions.

A. Timed States

A timed state consists of the untimed state, which is the set of enabled rules and the Boolean state, and the timing information which is represented by a set of *active clocks*. An active clock is created whenever a rule becomes enabled, and eliminated when the rule fires. After a firing sequence is executed, there is an active clock for every rule that is enabled by the execution of the firing sequence. The set of possible timing assignments to the sequence determines the set of possible ages that the active clocks can have. This set of ages essentially represents the timed state of the specification at the end of the firing sequence. Therefore, two firing sequences can be said to lead to the same timed state if they result in the same set of enabled rules, and the sets of possible ages for the active clocks resulting from the two sequences are the same.

This representation of equivalence classes leads directly to the geometric region method of representing time first introduced in [24], where the set of possible clock ages is represented by a set of inequalities. However, as described above, it suffers from an explosion in the number of geometric regions per untimed state when the specification is highly concurrent. This is due to the way the equivalence class is defined. Since a valid timing assignment must be monotonically increasing, sequences that have concurrent rules firing in different orders always result in different equivalence classes since the relative clock ages must reflect the firing order. This means that there is at least one region generated for each firing order that can lead to a given untimed state. This "region splitting" problem results in very poor performance for geometric region-based algorithms for concurrent examples.

The POSET algorithm uses a different method of defining equivalence classes, which significantly reduces this problem. When the POSET method is used, regions are generated based on the causality in the sequence. A firing of event e is causal to a firing of event f if the firing time of e controls the firing time of f. More formally:

Definition III.1: An event firing σ_i is *causal* to an event firing $\sigma_j \in \sigma$ if $\sigma_i = E_c(\sigma_{j-1}, \sigma)$.

Intuitively, this means σ_i is causal to σ_j if the firing σ_i enables the rule whose firing makes σ_j fireable, and thus controls the firing time of σ_j. In [15], we proved an inequality on timed Petri nets that implies the following inequality for TEL structure specifications where all Boolean expressions are **true**: if event firing σ_i is causal to event firing σ_j in σ, then $\tau(\sigma_i) + l(\sigma_{j-1}) \leq \tau(\sigma_j) \leq \tau(\sigma_i) + u(\sigma_{j-1})$ is true for all valid timing assignments τ to σ. These inequalities follow from the fact that a rule must fire

at some time between l and u time units after it becomes enabled. Since it is the firing of the causal event that enables the rule, the rule fires between l and u time units after its causal event fires. However, this inequality does not mean that for a given firing sequence σ there is always a timing assignment that allows the firing time of σ_i to reach all values within the range of the inequality. For the purposes of the algorithm, we would like to be able to create regions that contain the entire range. Therefore, it is necessary to look at more than one firing sequence at a time and show that there is always *some* firing sequence that allows any given value in the range to be assigned as a firing time.

B. Reorderings

The concept of a *valid reordering* of a firing sequence is defined in [15]. A valid reordering of a firing sequence is a change in the firing order that does not change the causality of the sequence and conforms to all of the requirements in Definition II.5. We proved in [15] that for specifications without conflict, it is always possible to create a reordering of a firing sequence so that there is a valid timing assignment where the separation between σ_i and σ_j reaches either of the bounds in the inequality above. For specifications with conflict, the bounds are not always reachable, since restrictions must be made to the reordering to ensure that conflicts are not resolved differently in the two sequences. In order to make timing assignments consistent with this restriction, rules with nonempty choice sets must be given timing assignments consistent with the *current* firing sequence. These properties are used to modify the standard geometric region algorithm so that it is not forced to produce at least one region for each firing order. Instead, regions are produced that contain timing assignments for all of the possible valid reorderings of the firing sequence.

In order to ensure that a reordering is valid, it must meet certain restrictions, which are described in [15] for TEL structures where all Boolean expressions are **true**. The reordering restrictions described in [15] are essentially as follows.

1) An event firing σ_i cannot be reordered to occur after a rule firing σ_j if $L(\sigma_i)$ is the enabling event of the rule firing in $L(\sigma_j)$.
2) A rule firing σ_i cannot be reordered to occur after an event firing σ_j if the firing of $L(\sigma_i)$ is one of the rule firings needed to fire $L(\sigma_j)$.
3) Since we do not want to change the causality, the rule preceding each event firing is the same in the original and reordered sequences.
4) Rule firings that have conflicting enabling events cannot be reordered since this may cause a different choice to be made between conflicting events.

These conditions ensure that the reordered sequence has a valid firing order, the causality remains unchanged, and the same choices between conflicting events are made in both sequences.

In order to modify the result from [15] to work on TEL structures with Boolean expressions, we need to determine the additional reordering conditions that are necessary to preserve causality in the sequence and ensure that any reordered sequence conforms to the requirements in Definition II.5. When there are no Boolean expressions, any reordering of the sequence where the rule firing immediately preceding each event firing does not change preserves the causality in the sequence. The last rule firing before an event σ_i fires is always the causal rule of σ_i by definition. If all Boolean expressions are **true**, then the enabling event for the rule σ_{i-1} is always the causal event for event firing σ_i. With Boolean expressions, this is not the case. Another event firing may have caused the rule that fires in σ_{i-1} to become enabled. A firing sequence σ must not be reordered in a way that changes the identity of this event. Additionally, due to the Boolean expressions, it is more difficult to ensure that the reordered sequence is a valid firing sequence according to Definition II.5. The sequence must only be reordered in a way that preserves the property that a rule's Boolean expression is always true at the time it fires.

If arbitrary Boolean expressions are allowed, determining which reorderings can be made without changing the causality or producing an invalid sequence is a difficult problem. When arbitrary Boolean expressions are included, the ability to change the firing order of σ_i and σ_j can depend on whether the location of another event σ_k has been changed. For example, consider the Boolean expression $a \wedge (b \vee c)$ on a rule r, where the enabled event is $f+$. Suppose that in the original firing sequence a, b, and c are all true when $f+$ fires. Either $b+$ or $c+$ could be reordered to occur after $f+$ because doing so would still allow the rule to be enabled when $f+$ fires. However, once the decision has been made to reorder $b+$ after $f+$, $c+$ cannot be reordered after $f+$, since r is not enabled when $f+$ fires if *both* $b+$ and $c+$ are reordered to occur after $f+$. Since reordering decisions are no longer independent, it is difficult to examine all possible reorderings at once because a reordering of one event may exclude the reordering of another event.

When each Boolean expression is restricted to be only purely conjunctive or purely disjunctive (with complemented literals), additional reordering restrictions for level expressions can be developed. For any firing $\sigma_i \in \sigma$:

1) if $L(\sigma_i)$ is a rule where $b = \textbf{true}$, there are no additional restrictions;
2) if $L(\sigma_i)$ is an event, there are no additional restrictions;
3) if $L(\sigma_i)$ is a disabling rule, no event that would disable $L(\sigma_i)$ can be moved before σ_i;
4) If $L(\sigma_i)$ is a nondisabling rule and $\sigma_j = E_c(\sigma_i, \sigma)$, no event that would prevent the firing of σ_j from enabling σ_i can be moved before σ_j;
5) if $L(\sigma_i)$ is a rule with a conjunctive (**and**) expression and σ_j is the causal event of σ_i, then:
   a) the enabling event of $L(\sigma_i)$ cannot be moved to occur after σ_j;
   b) no context firing can be moved to occur after σ_j;
6) if $L(\sigma_i)$ is a rule with a disjunctive (**or**) expression and σ_j is the causal event to σ_i, then:
   a) the enabling event of $L(\sigma_i)$ cannot be moved to occur after σ_j;
   b) no firing which causes the **or** expression to become true can be moved before σ_j;
   c) if $L(\sigma_j)$ is the enabling event of $L(\sigma_i)$, then no event can be reordered to occur after σ_j if it would cause the **or** expression to be false when σ_j fires.

The first four conditions apply equally to **and** and **or** expressions. Obviously, if there is no Boolean expression, then the old reordering restrictions that do not consider them are sufficient. If the firing is an event, there are no additional restrictions since all of the added conditions in the previous section concern rules. If a rule is disabling, a reordering should not cause a rule to become disabled if it does not do so in the original sequence. For example, if a disabling rule $e+ \rightarrow f+$ has a Boolean expression $a \wedge b$, a firing of $a-$ cannot be reordered to occur before the rule firing. If the rule is nondisabling, the restriction is needed that no event prevents the rule's causal event from enabling it. For example, consider the nondisabling rule $e+ \rightarrow f+$, which has Boolean expression $a \wedge b$, and causal event $e+$. If $e+$ is causal, then $a \wedge b$ is true when it fires. No firing of $a-$ can be moved to fire before $e+$ since it would not allow the firing of $e+$ to enable the rule.

There are specific additional restrictions for rules with **and**'s and **or**'s. If there is an **and** expression, then a reordering may change the causality or cause the new sequence to be invalid if some signal firing is moved later in the sequence. The restriction prevents a sequence from being created where the **and** expression is not true when the rule fires. Since no event firing that affects the expression may be moved after the causal event, it also ensures that the causal event for the rule firing remains the same. For example, consider a rule $e+ \rightarrow f+$, which has a Boolean expression $a \wedge b$, and assume that $b+$ is causal in the firing sequence. The fact that $b+$ is causal implies that there has been a firing of $e+$ and a firing of $a+$ somewhere in the sequence before $b+$. In a reordering, the firings of $e+$ and $a+$ are not allowed to be moved after $b+$ in the firing sequence.

With **or** expressions, three conditions are necessary. As with **and** expressions, the enabling event must not be reordered to occur after the causal event. The reordering also must ensure that the Boolean expression does not become true too *early*. If the reordering moves an event firing that satisfies the **or** to occur before the causal event then the causality changes. Therefore, this is not allowed. For example, consider a rule $e+ \rightarrow f+$, which has a Boolean expression $a \vee b$, and assume that $b+$ is causal in the firing sequence. The firing of $e+$ cannot move after $b+$ just like in the **and** expression. However, no firing of $a+$ can be allowed to move before $b+$, unlike in the **and** expression. If $a+$ fires first, then it is the causal event. The final condition ensures that the **or** expression is satisfied when the rule fires. If the enabling event is the causal event, the firing that satisfied the **or** expression cannot be moved after the firing of the enabling event. Arbitrary Boolean expressions require combinations of these requirements, which could be defined but would be difficult to implement in an algorithm that is building geometric regions. The next section describes how these reordering conditions are used to build geometric regions, which represent timing assignments to reorderings of the firing sequence.

IV. Timed State Space Exploration

Circuits specified as TEL structures are verified by using a depth-first search to find all of the states allowed by the specification. As firing sequences are explored, the current state after each firing is compared with all previously encountered states to determine if it has been seen before. In order to do this search, the algorithm needs to keep track of a set of rules whose enabling events have fired R_m, the Boolean state that results from the current firing sequence s_c, and the set of enabled rules R_{en}. The triple $R_m \times s_c \times R_{en}$ defines an *untimed state* since it indicates which rules are enabled but says nothing about timing. In order to determine which rules in R_{en} are satisfied, timing information (TI) is needed. A *timed state* is defined to be $R_m \times s_c \times R_{en} \times \mathrm{TI}$. A timed state contains all the information necessary to compute the set of satisfied rules R_s. Only rules in R_s are allowed to fire and cause a transition to another state.

A. Geometric Regions

As mentioned earlier, timing information is represented with geometric regions. The minimum and maximum age differences of all the active clocks are stored in a constraint matrix M. Each entry m_{ij} in the matrix M has the value $\max(c_j - c_i)$, which is the maximum age difference of the clocks. A dummy clock c_0, whose age is always zero, is also included. The maximum age difference between c_i and c_0 (m_{0i}) is the maximum age of c_i, and the maximum age difference between c_0 and c_i (m_{i0}) is the negation of the minimum age of c_i. This constraint matrix represents a convex $|R_{en}|$ dimensional region. Each dimension corresponds to an unfired rule, and the age at which it fires can be anywhere within the space.

When an event fires and causes new rules to be added to R_{en}, the matrix needs to be updated to reflect the new timing information. Information about the newly enabled rules must be added to the constraint matrix and information about rules that are no longer in R_{en} must be removed. The main operation used to do this is *recanonicalization*. Recanonicalization takes a matrix M where some of the m_{ij}'s are greater than $\max(c_j - c_i)$ and produces a matrix where all the m_{ij}'s have their maximum allowed value. The assignment of the m_{ij}'s so that they all have their maximum value is always unique; the algorithm can determine when a given region is equivalent to or contained in a region that has been seen before. Recanonicalization is, essentially, the all pairs' shortest path problem and can be done in $O(n^2)$ time with Floyd's algorithm [29].

Since our semantics consist of rule and event firings, rules fire independently of events and timing information is updated whenever a rule fires. In the algorithm, a rule can always fire when it is satisfied. The firing of a rule, however, does not always correspond to the firing of an actual event. An event only fires when a sufficient set of the rules enabling it has fired. As rules fire, they are projected out of the constraint matrix, removed from R_m, R_{en}, and R_s, and added to a new set of "fired" rules R_f. Since they have fired, timing information about them is no longer needed, but the fact that they have fired must be recorded. The set R_f is part of the timing information and therefore part of the timed state. When a set of rules sufficient to enable an event e are in R_f, e can fire.

B. POSET Timing—Updating the State

During the depth-first search, the algorithm calculates the satisfied set R_s from each timed state. It then chooses a rule from R_s to fire, places the rest of the rules in R_s on the stack, and calls a function that returns the timed state that results from firing the

rule. If the new timed state has been seen before, the algorithm pops an unexplored timed state off the stack and continues the search. If there are no more unexplored states on the stack, the algorithm has completed.

The POSET algorithm, used to reduce the state explosion problem, creates geometric regions based on partially ordered sets of events rather than linear sequences. The partial order is defined by the reordering restrictions described above. If two firing sequences are valid reorderings of each other, then they have the same partial order. Regions represent all possible age relationships between the enabled rules and can be generated by all firing sequences that have the same partial order as the firing sequence currently being explored. This prevents additional regions from being added for different sequences of event firings that lead to the same untimed state. POSET timing results in a compression of the state space into fewer and larger geometric regions that, taken together, contain the same region in space as the set of regions generated by the standard geometric technique. Therefore, all properties of the system that can be verified with the standard geometric technique can be verified with the POSET algorithm.

Fig. 5 shows the procedure for updating the timed state using the POSET timing technique. The algorithm does a depth-first search of the timed state space, finding all the timed states that are reachable. It first initializes all of the elements of the timed state. The set R_m is set to R_0, the set of initially marked rules in the TEL structure. The current state is set to the initial state of the TEL structure. The R_{en} set is created by including all marked rules whose Boolean expressions are satisfied by the initial state. The timing information M is then initialized for all the enabled rules. All initially enabled rules have a minimum age of zero and a maximum age of the least upper bound among them. Their relative age differences are all set to zero. The algorithm then initializes R_f to $\emptyset$. After these steps, the algorithm has created the initial timed state. It combines all the elements of the timed state into a data structure TS and adds it to the state space Φ. In order to use the state space for synthesis, the algorithm also must store the set of possible transitions between states. This set is called Γ and is initially empty. After initializing Γ, the algorithm calls the function $find_timed_enabled$, which returns the set of rules that are currently allowed to fire. It goes through all of the enabled rules and adds those whose clocks meet their lower bounds to the list of rules that can fire. The algorithm has now initialized everything and is ready to begin the main loop.

```
Algorithm .1 (Update the timed state)
void update(TEL structure TEL ⟨N, s_0, A, E, R, R_0, #⟩ ,
            geometric region M, rule r = ⟨e, f, l, u, b⟩,
            rule set R_en, R_new, bool event_fired) {
  R_m = R_0;  s_c = s_0;
  R_en = {⟨e, f, l, u, b⟩ ∈ R_m : b(s_c)};
  M = initialize_timing(R_en, TEL);
  PM =initialize_POSET(R_en, TEL);
  R_f = ∅;
  timed_state TS = R_en × R_m × s_c × R_f × M;
  set_of_states Φ = {TS};
  set_of_transitions Γ = ∅;
  rule_list RL = find_timed_enabled(TS, TEL);
  bool done=false;
  while (¬done){
    event_fired = false;
    rule r = ⟨e, f, l, u, b⟩ = head(RL);
    push(⟨TS, PM⟩, tail(RL));
    R_old = R_en;  R_f = R_f ∪ r;
    R_m = R_m − r;  R_en = R_en − r;
    if (∀ r_i = ⟨e_i, f, l_i, u_i, b_i⟩ ∈ R : ((r_i ∈ R_f)∨
       (∃r_j = ⟨e_j, f, l_j, u_j, b_i⟩ ∈ R_f : e_i#e_j)))){
      event_fired = true;
      if (f = ⟨x_i+, m⟩) s_c[s_index(x_i)] = 1;
      else if (f = ⟨x_i−, m⟩) s_c[s_index(x_i)] = 0;
      R_used = {⟨e_i, f, l_i, u_i, b_i⟩ ∈ R_f};
      R_m = R_m − {r_j ∈ R : f ∈ choice_set(r_j)};
      R_m = R_m ∪ {⟨e_i, f_i, l_i, u_i, b_i⟩ ∈ R : e_i = f};
      R_f = R_f − {⟨e, f_i, l_i, u_i, b_i⟩ ∈ R_f : f_i = f};
      R_f = R_f − {r_j ∈ R : f ∈ choice_set(r_j)};
      R_en = R_en − {r_j ∈ R : f ∈ choice_set(r_j)};
      R_en = R_en ∪ {r_j ∈ R_m : b_i(s_c)};
      foreach (r_i = ⟨e_i, f_i, l_i, u_i, b_i⟩ ∈ R_en ∪ R_f  :
               r_i is disabling)
        if (¬b_i(s_c))
          if (fail_on_disable) return fail;
          else R_en = R_en − r_i;
    }
    M = update(r, R_used, PM, M, TEL, s_c, , R_en,
               R_en − R_old, event_fired);
    TS_old = TS;  TS = R_en × R_m × s_c × R_f × M;
    if (TS ∉ Φ)  then
      Φ = S ∪ {TS};  Γ = Γ ∪ {(TS_old, TS)};
      RL=find_timed_enabled(TS, TEL);
    else if (TS ∈ Φ) then
      Γ = Γ ∪ {(TS_old, TS)};
      if (stack is not empty) then
        ((TS, PM), AL)=pop();
      else done = true;
  }
  return (Φ, Γ);
}
```

Fig. 5. Procedure for updating the timed state.

The main loop of the algorithm continues until all of the reachable states have been found, a condition represented by the variable *done*. When the loop begins, the function removes the rule it is going to fire r from the front of the rule list [i.e., *head (RL)*] and places the rest of the rule list [*tail (AL)*], the timed state, and the POSET matrix on the stack. Next, it saves the current R_{en} set by assigning it to R_{old}. This is done so that the algorithm can determine which rules in R_{en} are newly added. It then adds r to the fired set since it is firing and removes it from R_m and R_{en} since it is no longer available to fire. Next, the algorithm checks if firing of r causes an event to fire. An event fires if all of the rules that enable it are either in R_f or have enabling events that conflict with the enabling event of a rule that is in R_f. If an event can fire, the algorithm updates the state vector using the *s_index* function to find the index of the signal that is changing state in the state vector. If a sequencing event fires, the state vector remains unchanged. Next, the algorithm updates the rule sets to reflect the firing of a new event. The R_{used} set is set to contain all of the rules that are used in the firing of f. These are the rules that enable f and are in the fired set R_f when f fires. The marked set R_m loses all rules that contain f in their choice sets, since they have lost their chance to fire. The marked set gains all rules that have f, the firing event, as their enabling event. The fired set loses all rules that enable f, and all rules that contain f in their choice sets. These rules are no longer usable since they have either been used or become unusable due the firing of an event in their choice sets. The enabled set is also updated: it loses all rules that contain f in their

choice sets and gains all rules in R_m, whose Boolean expressions are satisfied in the new state. The algorithm then checks for rules that have been disabled. If a disabling rule is in the enabled set and its Boolean expression is no longer true due to the firing of f, it has been disabled. This can result in two different outcomes. If the designer wishes to consider disablings failures, since they correspond to hazards on the inputs of gates, then at this point the algorithm returns a fail condition and ATACS generates a failure trace. If the designer does not want the algorithm to fail on a disabling, the offending rule is removed from the enabled set and the algorithm continues. After all the rule sets have been updated, the algorithm updates the constraint matrix M and POSET matrix PM. The details of this are discussed in the next section. Next, the old timed state is saved in TS_{old} and all of the sets are combined into the new timed state. The algorithm then checks to see if this new state is already in the state space. If it is not in the state space, the new state is added is added to Φ and a new transition from TS_{old} to TS is added to the transition set Γ. Then a new list of rules to fire is computed from the current state. If the current state is already in Φ, the algorithm removes a state, a POSET matrix, and rule list from the stack and continues the main loop. If the stack is empty, then there are no more new states to be found and the algorithm is completed.

C. POSET Timing—Updating the POSET

The method for updating the POSET matrix is based on causality. When doing analysis on TEL structures with all **true** Boolean expressions, if $r = \langle e_c, e, l, u, b\rangle$ is the causal rule to e, then the firing time of the event e_c controls the firing time of event e. However, when the Boolean expressions are not **true**, this is not the case. The event that determines the enabling time of a rule may be its enabling event or it may be some other event firing that causes its Boolean expression to become satisfied. For example, consider the TEL structure for the signal $out2+$ in Fig. 2 when $out2-$ has just fired. Suppose that $out1$ rises and then c rises. In this case, $c+$ is causal to the event $out2+$. Assuming that the rule does not become disabled, $out2+$ must rise between ten and 30 time units after c rises. This subsection describes how the algorithm maintains the POSET matrix, taking into account the complexities caused by the ability of any event to be causal to any rule through a Boolean expression.

For any given rule firing sequence, there is a well defined causal event for each event firing r. Each event firing e has a causal rule firing r_c. The event firing that controls the firing time of e is the causal event of r_c. The timing of this causal event determines the minimum and maximum firing time of e over all reorderings of the firing sequence. The purpose of the POSET matrix is to keep track of the time separations between event firing times that are allowed by a valid reordering of the firing sequence without forcing the timing behaviors represented by the geometric regions to conform to the total order of the firing sequence. This prevents a new region from being generated for every possible firing sequence leading to an untimed state and drastically reduces the size of the state space.

Fig. 6 shows the procedure for updating the constraint matrix and the POSET matrix. The POSET matrix is only updated if

```
Algorithm .2 (Update the POSET)
update(causal rule r_c = ⟨e_c, f_c, l_c, u_c, b_c⟩,
        used rule set R_used, POSET matrix PM,
        constraint matrix M, TEL ⟨N, s_0, A, E, R, C, , #⟩,
        state s_c, R_new, bool event_fired){
  if(event_fired) then {
    forall(e_i : e_i is represented in PM){
      PM[index(f)][index(e_i)] = ∞;
      PM[index(e_i)][index(f)] = ∞;
    }
    forall(e_i : e_i is represented PM){
      if(e_i = causal(r_c)) then {
        if(∀r = ⟨e, f_c, l, u, b⟩ ∈ R : choice_set(r) = ∅) then
          PM[index(f_c)][index(e_i)] = u_c;
        else PM[index(f_c)][index(e_i)] = M[0][index(r_c)];
      if (disable(e_i, f_c)) then
        PM[index(e_i)][index(f_c)] = 0;
      forall(r = ⟨e, f_c, l, u, b⟩ ∈ R_used){
        e_rc = causal(r);
        if(e_i = e ∧ PM[index(f_c)][index(e_i)] > −l) then
          PM[index(e_i)][index(f_c)] = −l;
        if(e_i = e_rc ∧ PM[index(e_f)][index(e_i)] > −l) then
          PM[index(e_rc)][index(e_i)] = −l;
        if(and_context(e_i, r) ∧
           PM[index(e_rc)][index(e_i)] > 0) then
            PM[index(e_rc)][index(e_i)] = 0;
        if(or_context(e_i, r) ∧
           PM[index(e_i)][index(e_rc)] > 0) then
            PM[index(e_i)][index(e_rc)] = 0;
      }
    }
    recanonicalize(PM);
    forall (e_i : e_i is represented in PM){
      if (¬∃r_i = ⟨e_i, f_i, l_i, u_i, b_i⟩ ∈ R_en ∧ ¬match(e_i, s_c))
      then
        project( PM, index(e_i));
    forall(r_i = ⟨e_i, f_i, l_i, u_i, b_i⟩ ∈ R_en) {
      M[index(r_i)][0] = 0;
      forall(r_j ∈ R_en){
        M[index(r_i)][index(r_j)] =
          PM[index(causal(r_i))][index(causal(r_j))];
      }
    }
  }
  project(M, index(r));
  forall(r_i = ⟨e_i, f_i, l_i, u_i, b_i⟩ ∈ R_en)
    if (r_i ∈ C)       M[0][index(r_i)] = ∞;
    else        M[0][index(r_i)] = u_i;
  recanonicalize(M);
}
```

Fig. 6. Procedure for updating the POSET matrix.

an event fires. Each entry in the POSET matrix represents the maximum time separation possible between two event firings over all possible valid reorderings of the firing sequence. When a new event f_c fires, entries must be added to store the separations between f_c and all of the other events represented in the matrix. The function first initializes all of the new entries in the matrix to infinity. A value of infinity means that there is no reordering restriction that applies to this event pair.

The rest of the algorithm checks the various reordering restrictions and changes the values in the matrix accordingly. For each event e_i in the POSET matrix, the algorithm first determines if e_i is the causal event to the causal rule r_c. If e_i is the causal event and the firing event is not enabled by any rules with a nonempty choice set, then its firing time determines the upper bound on the firing time of f_c over all valid reorderings. This

separation is thus set to the upper bound of the causal rule u_c. If the firing event f_c is enabled by a rule with a nonempty choice set, then the upper bound in the POSET matrix is set to the upper bound on the causal rule in the constraint matrix. This sets the upper bound on the firing time of f_c to be the latest allowable by the *current* firing sequence. Then the function checks if this event firing could disable a rule that enables the event in the POSET matrix that is currently being examined e_i. If it does, then f_c must always occur after e_i and their minimum separation is set to zero, indicating that f_c cannot occur before e_i.

The next step is to check all of the other reordering restrictions. Since the reordering restrictions are defined with respect to rule firings, the algorithm needs to apply the reordering restrictions to all of the rule firings that are used to fire the event f_c. First, the algorithm extracts the causal event for the rule that it is considering e_{rc}. In practice, it is simple to store the causal event of a rule when it becomes enabled. It then checks to see if e_i is an enabling event of r. If e_i is the enabling event of r, then the lower bound on r must be met for any valid reordering and the lower bound in the matrix is set to $-l$ if it is not already less than $-l$. The event e_i may also be the causal event of r, and this also implies the minimum separation between e_i and f_c is l. Next, the algorithm checks for events that are required for an expression associated with r to be satisfied. Any such events must fire before the causal event, and therefore the minimum separation between them and the causal event is set to zero. Note that an event can be considered *and_context* even if it is associated with an **or** expression. If the causal event of a rule with an **or** expression is its enabling event, then one other event is necessary in order for the **or** expression to be true when the rule becomes enabled. This event is *and_context* for the **or** rule. For events with **or** expressions, there is also an opposite restriction. Any events that would cause the value of the **or** expression to become true before the causal event fires must not be reordered to occur before the causal event. Therefore the maximum separation between e_i and e_{rc} is set to zero to ensure that e_{rc} cannot happen after e_i. These entries in the POSET matrix ensure that none of the timing assignments allowed violate the reordering restrictions.

After the new constraints are added, the matrix is recanonicalized, which tightens all of the separations down to the maximum allowed by the known constraints. Finally, any events that are no longer relevant to future behavior of the system are removed from the matrix by the *project* function. An event can no longer affect future behavior if it is not causal to any rule currently in the constraint matrix and the direction of the signal transition no longer matches the current state ($a+$ no longer matches the current state if a is low in the current state). The result is a POSET matrix that constrains the minimum and maximum separations between events to bounds that are implied by the causality in the firing sequence. The constraints computed in the POSET matrix can then be used to compute a new constraint matrix M. The minimum age of each rule is set to zero since information about minimums is already included in the POSET matrix. Next, the algorithm sets each entry in the constraint matrix, which represents age differences between rules, to the time separation between their causal events. Then the algorithm projects out the entry in the constraint matrix for the rule that is firing. Finally,

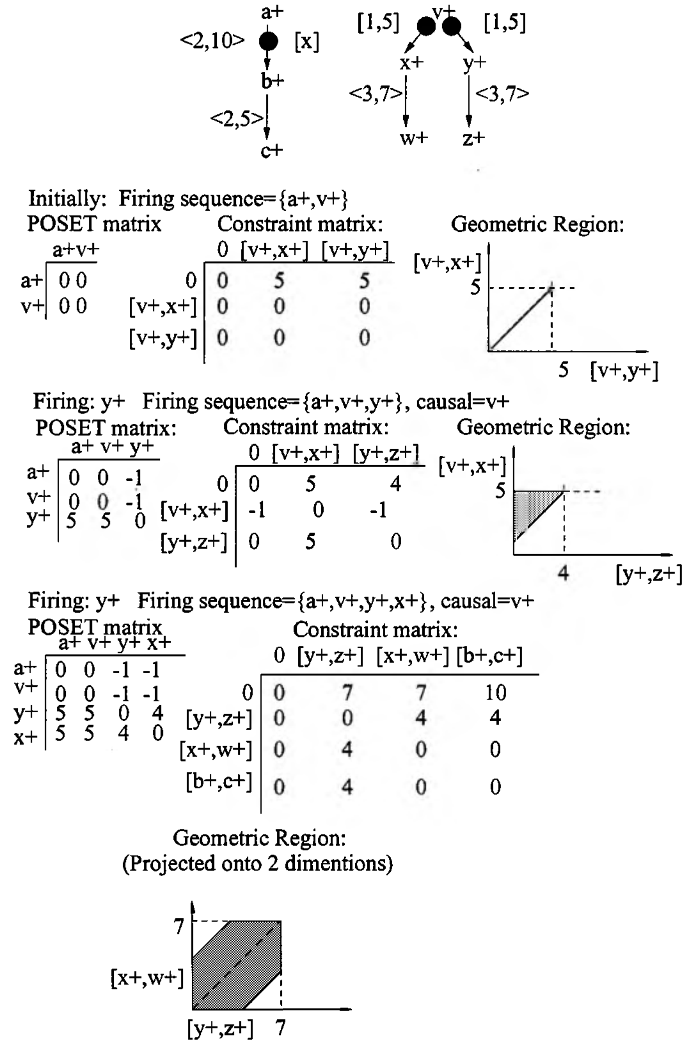

Fig. 7. Example of POSET algorithm.

the algorithm updates the constraint matrix. It sets the maximum age of each constraint rule in the matrix to infinity and the maximum age of all of the other rules in the matrix to their maximum possible age u_i. This allows time to advance as far as possible without causing any rule to exceed its maximum age. Since the upper bounds for constraint rules are set to infinity, they do not constrain the region, and adding constraint rules to a specification does not cause the generation of new regions.

This algorithm extends the benefits of POSET timing to specifications with level expressions. The additions that are necessary to support levels do not add significantly to computation time, since they simply consist of determining causality and context relationships. When TEL structures are limited to simple **and** or **or** terms, these relationships can be determined by checks that occur when a rule becomes enabled, and require very little computation time.

V. Example

Fig. 7 shows the application of the POSET algorithm to the TEL structure fragment shown at the top. The initial state of the TEL structure is indicated by its marking. The rules $[a+, b+]$, $[v+, x+]$, and $[v+, y+]$ are initially in R_m. The

all zero initial POSET matrix indicates that $a+$ and $v+$ have fired at the same time. The initial constraint matrix indicates that only two of the three marked rules are initially enabled, $[v+, x+]$ and $[v+, y+]$. The rule $[a+, b+]$ is not initially enabled since its Boolean expression $[x]$ is not satisfied in the current state. The initial geometric region shows that the two rules must have the same age and that their age cannot exceed five, which is the upper bound for both of them. Given this region, either rule can fire. In this example, rule $[v+, y+]$ is chosen to fire first. Since this is the only rule that enables event $y+$, $y+$ immediately fires, resulting in the firing sequence $a+$, $v+$, $y+$. The POSET matrix generated by this firing sequence shows that the separation between the firing of $v+$ and $y+$ is between one and five, as is the separation between the firing times of $a+$ and $y+$. The region constructed using this POSET matrix shows that the rule $[v+, x+]$ has a maximum age of five, while the rule $[y+, z+]$ has a maximum age of four. The region also requires that $[v+, x+]$ must be at least one time unit older than $[y+, z+]$. The benefits of the POSET algorithm are illustrated with the next firing $x+$. The POSET matrix created by the firing of $x+$ shows that $y+$ and $x+$ are allowed to fire in either order ($y+$ can fire up to four time units after $x+$ and $x+$ can fire up to four time units after $y+$). This occurs because there are no causal relationships that require the firing order of $y+$ and $x+$ to enforce in the POSET matrix. Since $x+$ and $y+$ are concurrent events, and changing their firing order does not change any causality relationships, a region is constructed that includes both the current firing sequence and another firing sequence, where $x+$ fires before $y+$. The line drawn through the shaded region shows the two regions that would be generated using the standard geometric region algorithm. The POSET algorithm allows one region to be generated, and thus reduces the number of regions explored during state space exploration.

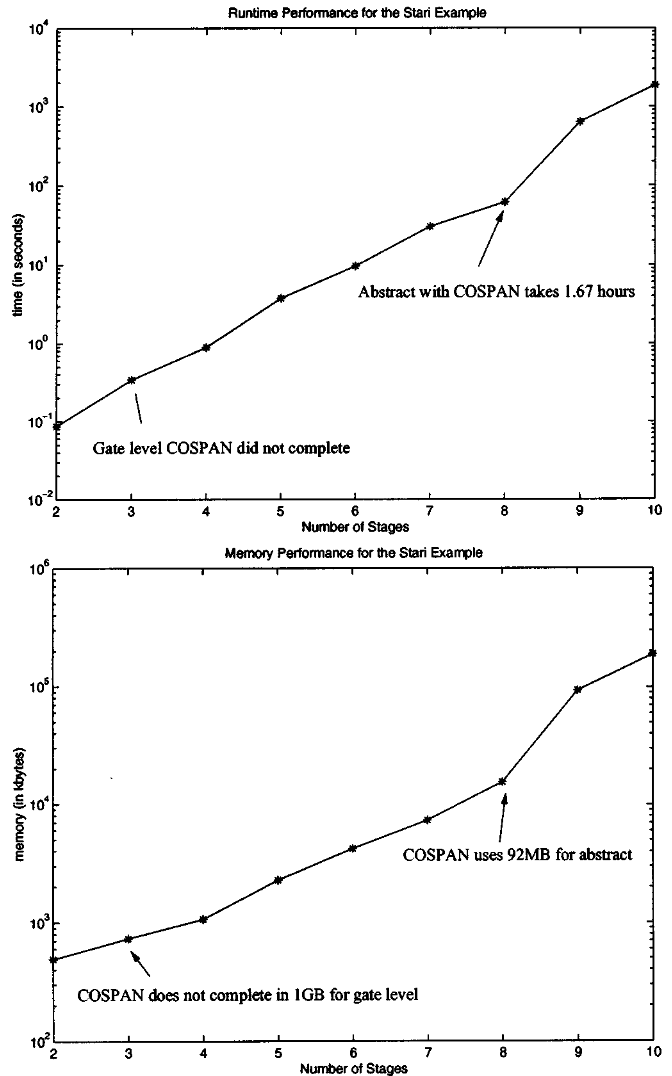

Fig. 8. Stari results with POSETs and with COSPAN.

VI. Circuit Analysis

The POSET algorithm applied to TEL structures is implemented in the CAD tool ATACS and has been applied to several examples. These examples, which come from both the synchronous and asynchronous domain, all depend on timing behavior for correctness. The reduction in state space size provided by the POSET algorithm makes timing verification tractable for circuits of interesting complexity.

A. Asynchronous Circuit Verification

The first example is a self-timed at receiver's input (STARI) communication circuit, described in detail in [33] and [34]. The STARI circuit is used to communicate between two synchronous systems that are operating at the same clock frequency π, but are out-of-phase due to clock skew which can vary from zero to *skew*. The environment of this circuit is composed of a *clk* process, a transmitter, and a receiver. The STARI circuit is composed of a number of first-in first-out (FIFO) stages built from two C-elements and one NOR-gate per stage, which each have a delay of l to u. There are two properties that need to be verified: 1) each data value output by the transmitter must be inserted into the FIFO before the next one is output [i.e., $ack(1)-$ precedes $x(0).t-$ and $x(0).f-$]; and 2) a new data value must be output by the FIFO before each acknowledgment from the receiver [i.e., $x(n).t+$ or $x(n).f+$ precedes $ack(n+1)-$] [35]. To guarantee the second property, it is necessary to initialize the FIFO to be approximately half full [34]. In addition to these two properties, we also verified that every gate is hazard-free (i.e., once a gate is enabled, it cannot be disabled until it has fired).

There have been two nice proofs of STARI's correctness [34], [36], but they have been on abstract models. Fig. 8 shows the runtime and memory results of running the POSET algorithm on STARI. These results are compared to those from [35], where COSPAN is used to verify STARI. Arrows in the figure indicate the performance of COSPAN reported in [35]. In [35], the authors state that COSPAN, which uses the unit-cube (or region) technique for timing verification [37], ran out of memory attempting to verify a three-stage gate-level version of STARI on a machine with 1 GB of memory. The paper goes on to describe an abstract model of STARI, for which they could verify eight stages in 92.4 MB of memory and 1.67 h. We first verified STARI at the gate-level with delays from [35] (i.e., $\pi = 12$,

skew $= 1$, $l = 1$, and $u = 2$). Using POSET timing, we can verify a three-stage STARI in 0.74 MB in only 0.40 s. For an eight-stage STARI, the verification took 11 MB and only 55 s. In fact, POSET timing could verify ten stages in 124 MB of memory in less than 20 min. This shows a nice improvement over the abstraction method and a dramatic improvement over the gate-level verification in COSPAN. For ten stages, POSET timing found 14 531 untimed states and only needed 14 859 geometric regions to describe the timed state space. This represents a ratio of only 1.02 geometric regions per untimed state.

Finally, the complexity of POSET timing is relatively independent of the timing bounds used. We also ran our experiments using $l = 97$ and $u = 201$, *skew* $= 101$, and $\pi = 1193$, which found more untimed states. With $l = 102$, we found less untimed states. Both cases with higher precision delay numbers had comparable performance to the one with lower precision delay numbers. This shows that higher precision timing bounds can be efficiently verified and can lead to different behaviors. It would not be possible to use this level of precision with a discrete-time or unit-cube-based technique, since the number of states would explode with such large numbers.

Recently, there have been a couple of new approaches taken to verify the STARI circuit. In [38], a discrete-time BDD-based approach implemented in KRONOS is used to verify 17 stages of STARI. While this result is very good, the timing bounds used in the verification have to be very small ([0,1] or [1,2]) to control the complexity. This approach would quickly explode if more significant digits are needed. Previously, we have shown in [15] that our approach substantially outperforms KRONOS for their highly concurrent benchmarks [12]. In particular, we analyzed 512 stages of *alpha* while they could only do 18, and we analyzed 14 stages of *beta* while they could only do nine. In [39], a partial order approach is applied to the verification of 11 stages of STARI. The performance is extremely good completing the verification in under 1 s using less than 1 MB of memory. However, they do not find the entire state space, making it impossible to use for synthesis, and the amount of improvement demonstrated by the algorithm depends on the properties to be verified.

The next example comes from the Intel RAPPID design [1]. The key to the performance of the RAPPID design is a very efficient synchronization mechanism which is called the *tagunit*. One *tagunit* is shown in Fig. 9. The operation of this circuit is that it can receive a tag from one of seven other tag units ($Tagin_i$). If the instruction is ready (*InstRdy*) and the crossbar is ready (*XBRdy*), it tags out to one of seven other tag units ($TagOut_i$) depending on the length of the instruction ($Length_i$). In order to compare the performance of the new, level-based approach to the timed Petri-net-based algorithm we presented in [15], we converted the tag unit TEL structure to a timed Petri-net and applied the POSET algorithm from [15]. The level based tag unit requires less time and fewer regions. The POSET algorithm completes analysis on the level based tag unit in 13 s, using 1246 regions. The algorithm from [15] requires 4518 regions and 103 s to analyze the timed Petri net. The level-based specification produces nearly a four times improvement in region count and a ten times improvement in runtime. In this example, the improvement in runtime and region count comes entirely from the improvement in the specification method. When the geometric

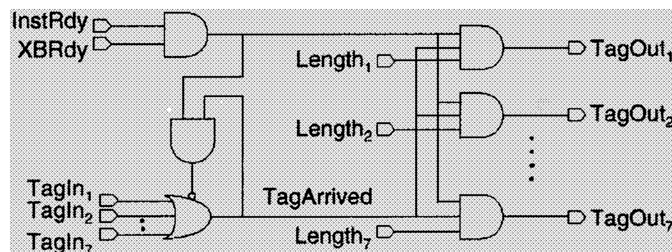

Fig. 9. Tag unit circuit.

algorithm without POSETS is applied to the level-based specification, the verification is completed in 13 s with a region count of 2089. The POSET algorithm does not produce a significant improvement over the standard algorithm on this example due to limited concurrency and extensive choice in the example.

In order to further examine the source of the improvement generated by applying the POSET algorithm to TEL structures, we created a TEL structure specification and a timed Petri-net specification of the high-performance FIFO element described by Molnar in [40]. This example is highly concurrent and choice-free. For a three-stage FIFO, the level-based specification without POSETS requires 23 540 regions and 121 s to complete. The same specification with POSETS requires 1405 regions and 5 s. The timed Petri-net specification with POSETS requires 4797 regions and 16 s. The timed Petri-net specification without POSETS does not complete. In this example, the POSET algorithm produces nearly a 20 times improvement when applied to the TEL structure-based specification, while moving from the timed Petri-net specification to the TEL structure specification produces approximately a four times improvement. In this example, the POSET algorithm produces a more significant improvement than the TEL structure specification method. Our experience in analyzing these and other examples has indicated that the POSET algorithm produces the largest improvements when the specification is highly concurrent and choice is absent or limited. When a specification is dominated by choice behavior, the largest improvement is gained by moving from an event-based to a level-based specification.

B. Synchronous Circuit Verification

ATACS has also been used to analyze several circuits from the guTS integer microprocessor designed at IBM's Austin Research Laboratory [2]. The purpose of this design is to demonstrate the performance gains that can be achieved by using aggressive circuit design. It is implemented in a $0.25\ \mu$m CMOS process available in 1997. The high performance of the circuit is a result of the circuit design, which is done is a dynamic circuit style known as delayed-reset domino [41], [42]. Although TEL structures and the POSET algorithm were originally developed to analyze asynchronous circuits, they are well suited to the analysis of delayed-reset domino circuits. The microprocessor contains a set of macros, which operate synchronously. A delayed-reset domino macro consists of a number of levels of dynamic gates, each of which receives inputs from preceding layers. Standard domino gates use a common clock that acts as

a timing reference. In a delayed-reset design, each level of dynamic gates receives its own, precisely timed clock, which is generated by a buffer chain within the macro. The local clocks travel through the logic along with the data, a reset wave preceding each computation wave. This technique allows approximately one-half cycle for each gate to reset and one-half cycle for each gate to evaluate. The cycle time for a delayed-reset domino macro is set by adding the necessary precharge and evaluate times for a single gate. If multiple gates operate on the same precharge signal, cycle time is set by adding the evaluate delay through all the stages to the precharge delay. Due to the overlapping of the precharge and evaluate phases, the delayed-reset domino approach significantly increases the amount of dynamic-logic that can be placed in a macro at a given clock frequency.

The delayed-reset domino gates used in the guTS processor lack the "foot" device that is included in a standard domino gate. The purpose of this device is to turn off the gates' pulldown stack during the precharge phase. Removing this device allows the gate to switch 5% to 15% faster. Alternatively, the gate can compute a more complex logic function using the same transistor stack height [41]. In order to remove this transistor, it is necessary to ensure that the evaluate logic is not on during the precharge phase. This is the case if all inputs to the gate are guaranteed to be low during the precharge phase. To meet this requirement, the inputs to the macro must be pulsed. Combined with the requirement that the inputs to each gate remain stable high long enough to switch the dynamic node, this results in a two-sided timing-verification problem, which is unusual for a synchronous design.

In the guTS processor, the macro level timing verification is done using extensive SPICE level circuit simulation [43]. After the delay behavior of the macros is characterized by designers in SPICE, it is incorporated into a chip-level timing model for chip-level static timing verification. This was a successful approach for this processor since it worked first in silicon. However, in order to ensure the correctness of the processor over all variations in delay, large amounts of delay margin are included in the design of the macros. If it is possible to formally verify the macros, then less margin is necessary to have confidence in the processor's correctness, which can result in higher performance. The timing constraints that need to be checked in the delayed reset domino macros are very similar to the correctness constraints necessary for asynchronous circuits, and the delayed reset domino circuits are quite similar to asynchronous circuits. Therefore, an asynchronous timing verification tool is a natural choice to be used for formal verification of the macros.

1) Verification of Gate Level Models: The asynchronous timing verification tool ATACS is used to verify several of the macros from the guTS processor. The first circuit is a combined multiplexor and latch (MLE). This circuit is small enough verify at the gate level and is shown in Fig. 10. The goal with this circuit is to verify that the timing specification, which is supplied with the circuit, indeed guarantees that the circuit works correctly. The timing specification describes the timing requirements, which must be met by any circuit communicating with the MLE. It is derived from SPICE level simulation and the circuit designers knowledge of how the circuit works. The

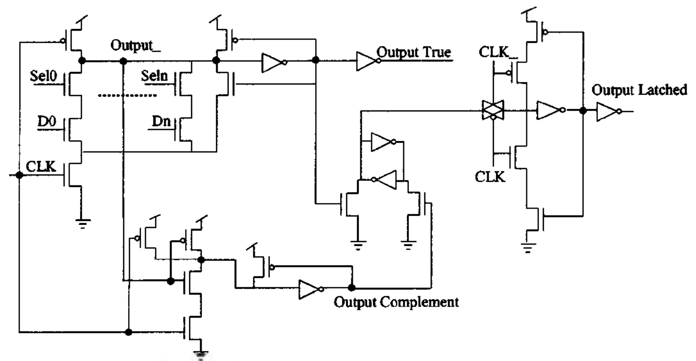

Fig. 10. MLE circuit.

timing specifications are also used as the basis for chip-level static timing analysis. In order to ensure that the chip-level static timing analysis is modeling all timing behavior, each macro needs to be formally verified in the environment described by the timing specification. ATACS verifies the MLE circuit in a few seconds on a 400-MHz Pentium II.

The MLE circuit contains both static and dynamic gates. The inputs to static gates are allowed to be unstable since this does not immediately cause a failure. However, if a glitch on the output of a static gate propagates to the input of a dynamic gate, it can cause a failure. In the MLE circuit, the gate driving the signal "output complement" is static. In every cycle where "output complement" does not fall, there is a glitch on its inputs. At the end of the precharge phase, the signal "Output_" is always high and it feeds one of the inputs to the static gate. When the clock rises, "output complement" always begins to fall. However, the signal "Output_" falls later in the clock cycle if the selected data value is high. When "Output_" falls, one of the inputs to the static gate is driven low and "output complement" rises again, producing a glitch. ATACS detects this glitch and determines that it cannot propagate to the output of the circuit.

The next circuit is a dynamic programmable logic array (PLA) that is used in the processor's control circuitry. Dynamic PLAs are easy to generate automatically and have predictable area and delay. In order to make the PLAs fast, they are controlled using self-resetting circuitry. An example of the control circuitry is shown in Fig. 11. The circuit uses a very aggressive technique to determine when its inputs are valid. The inputs are presented to the circuit dual-rail. When the inputs are valid, the sensor transistors are turned on. These transistors are all connected to a single node $n1$, which has been precharged high. The sensor transistors are sized so that one of them must be turned on for each input in order for $n1$ to discharge quickly. However, if one input arrives much earlier than the rest, eventually its single sensor transistor can discharge $n1$, erroneously causing the PLA to begin evaluating early. This completion detection circuit is highly timing dependent and only works if the inputs are guaranteed to arrive within a narrow time interval. After the falling edge of $n1$ propagates through four inverters, the node $n2$ falls. When this node falls, transistor $p1$ is turned on which raises node $n1$, resetting the completion detection circuit. The line "and plane control" is used to gate transistors, which determine if the and-plane of the PLA is in

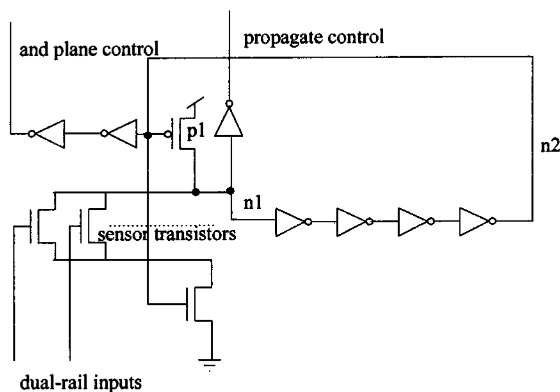

Fig. 11. PLA control.

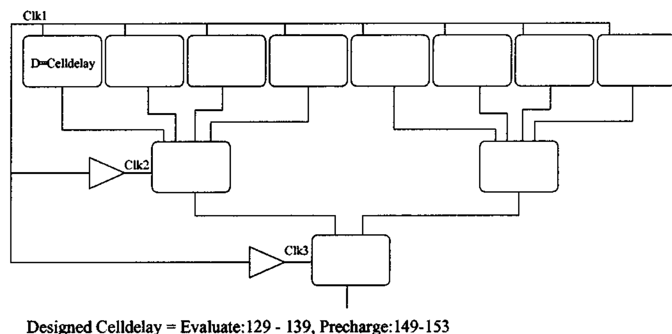

Fig. 12. Model for the compare unit.

precharge or evaluate mode. The line "propagate control" is used in a similar manner to control whether the output of the and-plane can propagate to the or-plane of the PLA, which is not shown. This control circuitry is essentially asynchronous. Self-resetting circuits are difficult for static tools to handle, since they often assume that a transition on an input causes only a single transition on an output. ATACS is able to verify the circuit using the designed delays in a few seconds.

2) Verification of Abstracted Models: The next circuit is a compare unit for two 64-b quantities. It consists of three stages of delayed-reset domino logic. The logic in each stage is exactly the same. A stage consists of a set of blocks that produce an output, which indicates whether its two 4-b inputs are equal. To do a 64-b compare, a tree structure is used where the first stage has 16 logic blocks, the second stage has four logic blocks, and the final stage has one logic block. Unlike the previous two examples, this circuit is too large for ATACS to verify it, using a representation derived directly from its transistor level schematic. In order to verify these circuits, we applied several conservative abstraction techniques by hand to reduce the complexity of the design. In the future, we plan to formalize and automate the techniques described here.

In the compare unit, reducing the bit width sufficiently reduces the complexity of the circut. It is not necessary to model each of the 64 b entering the compare unit. Each block in the first level of logic is modeled as a gate that waits for a single input and produces its output some variable amount of time later. Variability in input signal arrival times is accounted for by putting an independent delay range on the arrival time of the abstracted input signal for each of the blocks in the first level of logic. When this signal rises in the abstracted model, it is equivalent to all eight input bits to a block becoming stable in the actual circuit. Additionally, since the timing behavior of each block is the same, the number of input blocks can be reduced from 16 to 8 without effecting the timing behavior of the circuit. Fig. 12 shows the structure of the model. Each block is represented as a TEL structure, which raises its output signal 129 to 139 time units after the block receives all of its inputs, and lowers its output 149 to 153 time units after its local clock falls. A global clock, which controls the transition times of the local clocks, is also modeled but not shown. It takes less than 5 s to explore the state-space of this model using the POSET state space algorithm on a 400 MHz Pentium II. This circuit example also demonstrates the advantages of the level-based specification. The iteration time provided by the POSET algorithm makes it reasonable to iteratively adjust the celldelay values, global clock speed, and local clock timings to determine the working ranges of the circuit under a variety of assumptions. The circuit verifies for global clock cycles up to 100 ps less than the clock cycle necessary for correct operation in the gigahertz processor.

Since state-space exploration is an exponential problem, large specifications can only be verified at a high level of abstraction. This is illustrated by the verification of the 64-b adder portion of the multifunction fixed point unit (MFXU). This unit computes the results of the add, subtract, and compare instructions for the processor. The core of the unit is the 64-b parallel prefix adder design presented in [44], which is based on the algorithm described in [45]. The MFXU adder contains five stages of delayed-reset domino logic. The first stage contains a true/complement mux, stages two through four compute the propagate and generate signals for the adder, and the fifth stage implements a large mux, which merges many different signals. Each block contains a few domino gates, which can vary in delay. Attempts to verify this circuit at the gate level quickly use more than half of a gigabyte of memory and do not complete. However, a conservative abstraction of the MFXU verifies in ATACS using the POSET algorithm in about 2 min.

The structure of the MFXU abstraction is shown in Fig. 13. There are two steps involved in creating the conservative abstraction of the MFXU. The first is to reduce the complexity of each block by lumping the delay ranges for all of the different gates into one delay range, which represents the minimum and maximum time difference between the block receiving all of its inputs and generating all of its outputs. For example, suppose a block contains two domino gates d_1, which takes 100 ps to evaluate and d_2, which takes 150 ps to evaluate. It is conservative to make a model for the block where the minimum evaluate time for the block is 100 ps and the maximum evaluate time for the block is 150 ps. This abstraction does not capture the gate-level behavior that one output of the block is available after 100 ps and the other is available after 150 ps, but if a circuit verifies using the abstraction, its actual behavior verifies also. When an abstraction like this is made for the precharge phase and the evaluate phase of each block, then the number of blocks is decreased. The goal is to reduce the number of blocks without hiding any interesting block interactions. This is done by analyzing a 32-b-wide slice of the design. Since each block

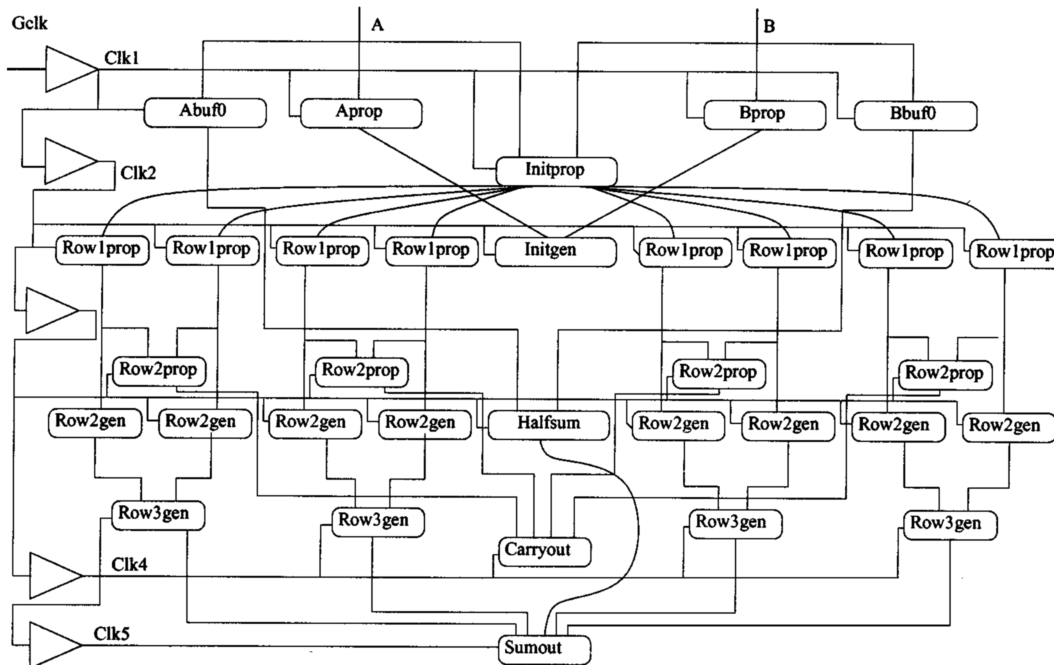

Fig. 13. MFXU structure.

operates on 4 b of input, this corresponds to a model that is eight blocks wide at its input. This model is large enough to include all of the types of interblock relationships of the larger design and is small enough to verify quickly.

This is done by starting at the last stage and working toward the first. Every block in the last stage is included. Then, for every block in the last stage, at least two instances of each type of block that provides inputs to the last stage are included in the fourth stage. In this case, four instances of the *row3gen* block which feeds *sumout* block in the fifth stage are included. Only one instance of the *halfsum* block is included, since there is only one *halfsum* block in the complete circuit. This process is then repeated for the fourth through first stages. The resulting model represents a conservative model of the possible timing relationships in the circuit, and is small enough to verify quickly.

The circuit, abstracted in this way, verifies at its intended clock speed. Therefore, any gate-level timing relationships that are missed by the abstraction are not necessary in order for the circuit to run at the specified speed. If this is not the case, then the blocks on the failure path can be specified in more detail. Although this increases verification time, it should not make the problem intractable since the additional detail is limited to a few blocks. Even if the abstracted version of this circuit is quite large and has complex timing relationships, which provide many possibilities for error. Formal verification gives confidence that all of the timing behaviors have been considered. Currently, ATACS does not have an automated method for generating circuit abstractions, and the abstraction described for this example is done manually. It may be possible to adapt techniques from [46] to develop an automated method for abstracting blocks of domino gates.

The final circuit, shown in Fig. 14, is an arithmetic circuit used in the integer execution unit. It is of moderate complexity and therefore can be used to test the accuracy of an abstracted model versus a gate-level model. The gate-level model is still somewhat abstract in that it does not include the full 64-b datapath, but each instance of a block is described at the gate level. The results on this macro indicate that the limiting factor in clock speed is the time that the inputs arrive to the macro, not gate-to-gate interactions inside the macro. Because of this, the maximum clock speeds allowed by the abstracted model and the gate-level model are the same. In order for a gate-level model to allow a circuit to verify at a higher clock speed than an abstracted model, there need to be instances of fast gates in one stage feeding slow gates in another block in the next stage. Such instances do not occur in this example.

VII. Conclusion and Future Work

Our results show that the POSET algorithm, when applied to TEL structures, can dramatically improve the efficiency of timing verification allowing larger, more concurrent timed systems to be verified. It does so without eliminating parts of the state space, so it does not limit the properties that can be verified. Due to the efficiency of the algorithm and the flexibility of TEL structures, ATACS is very effective for the verification of both synchronous and asynchronous circuits. Since ATACS is designed for asynchronous circuits, it can be used to verify many different circuit styles by varying the constraints that are checked. When circuit-level timing specifications can be verified, less margin is necessary in each circuit to ensure that the circuit works correctly, which can result in higher performance. ATACS does a complete state-space exploration. Therefore, its complexity is exponential and it is not practical to verify large circuits at the gate level. However, for most circuits, a higher level of abstraction is sufficient to verify that the circuit can run at the desired speed. If this is not the case, it is possible to locally specify more detail on paths that fail without causing a state

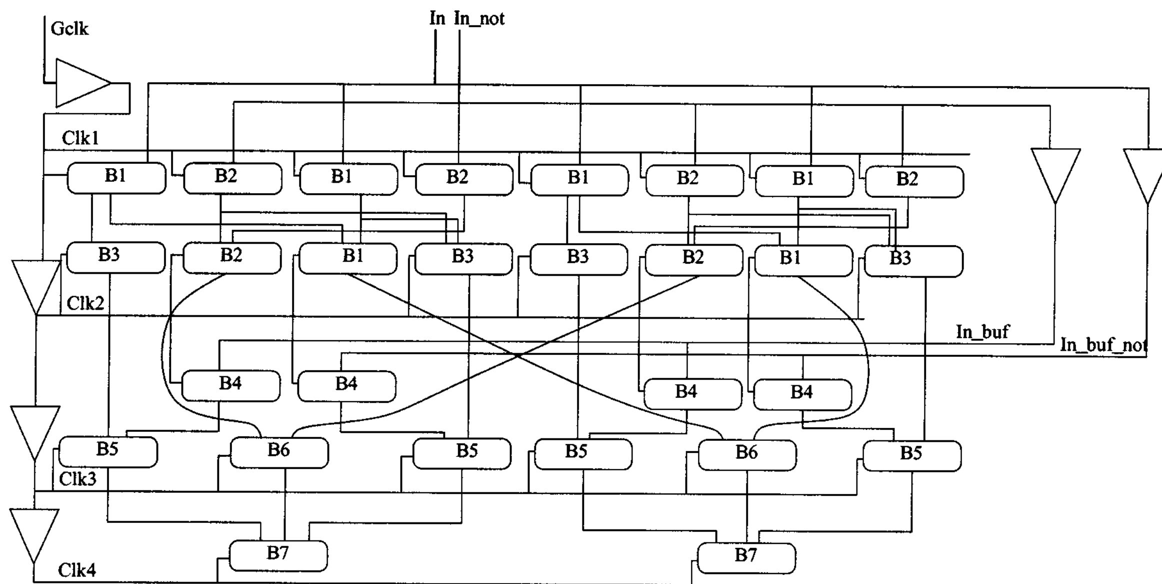

Fig. 14. CLZ circuit.

explosion. Most importantly, this paper shows how tools developed for asynchronous circuits can be useful to synchronous designers when they choose aggressive circuit styles.

In order to make this method practical for circuit designers, more work is needed to develop a more automated method of abstracting circuits and to develop a method of verifying circuits hierarchically. Additionally, all of the circuits described in this paper are completed and no failures are found by ATACS when designed delays are used. It would be interesting to study how ATACS can help designers determine which delay ranges result in correct circuits closer to the beginning of the design cycle, as well as how it can be used on early versions of circuits to find actual failures. Finally, we would like to explore how the synthesis capabilities of ATACS can be used to help automate the design of delayed-reset domino and self-resetting circuits.

Acknowledgment

The authors would like to thank all of the guTS design team at IBM, especially Dr. K. Nowka, for explaining the operation of the circuits used in guTS.

References

[1] S. Rotem, K. Stevens, R. Ginosar, P. Beerel, C. Myers, K. Yun, R. Kol, C. Dike, M. Roncken, and B. Agapiev, "RAPPID: An asynchronous instruction length decoder," in *Proc. Int. Symp. Advanced Res. Asynchronous Circuits Syst.*, Apr. 1999, pp. 60–70.

[2] H. P. Hofstee, S. H. Dhong, D. Meltzer, K. J. Nowka, J. A. Silberman, J. L. Burns, S. D. Posluszny, and O. Takahashi, "Designing for a gigahertz," *IEEE Micro*, pp. 66–74, May–June 1998.

[3] D. Van Campenhout, T. Mudge, and K. Sakallah, "Timing verification of sequential domino circuits," in *Int. Conf. Computer-Aided Design*, Nov. 1996, pp. 127–132.

[4] V. Narayanan, B. Chappel, and B. Fleischer, "Static timing analysis for self-resetting circuits," in *Int. Conf. Computer-Aided Design*, Nov. 1996, pp. 119–126.

[5] E. J. Shriver, D. H. Hall, N. Nassif, N. E. Raham, N. L. Rethman, G. Watt, and J. A. Farrell, "Timing verification of the 21 254: A 600 MHz full-custom microprocessor," in *Int. Conf. Computer Design*, Oct. 1998, pp. 96–103.

[6] P. Merlin and D. J. Faber, "Recoverability of communication protocols," *IEEE Trans. Commun.*, vol. COM-24, pp. 1036–1043, Sept. 1976.

[7] R. Alur, "Techniques for automatic verification of real-time systems," Ph.D. dissertation, Stanford Univ., Stanford, CA, 1991.

[8] T. Yoneda, A. Shibayama, B. Schlingloff, and E. M. Clarke, "Efficient verification of parallel real-time systems," in *Computer Aided Verification*, C. Courcoubetis, Ed. New York: Springer-Verlag, 1993, pp. 321–332.

[9] A. Semenov and A. Yakovlev, "Verification of asynchronous circuits using time Petri-net unfolding," in *Proc. ACM/IEEE Design Automat. Conf.*, 1996, pp. 59–63.

[10] E. Verlind, G. de Jong, and B. Lin, "Efficient partial enumeration for timing analysis of asynchronous systems," in *Proc. ACM/IEEE Design Automat. Conf.*, 1996, pp. 55–58.

[11] H. Hulgaard and Dept. of Comput. Sci., Univ. of Washington, "Timing analysis and verification of timed asynchronous circuits," Ph.D. dissertation, St. Louis, MO, 1995.

[12] M. Bozga, O. Maler, A. Pnueli, and S. Yovine, "Some progress in the symbolic verification of timed automata," in *Proc. Int. Conf. Computer-Aided Verification*, 1997, pp. 179–190.

[13] H. Zheng, "Specification and Compilation of Timed Systems," M.S. thesis, Univ. of Utah, Salt Lake City, 1998.

[14] W. Belluomini, "Algorithms for synthesis and verification of timed circuits and systems," Ph.D. dissertation, Univ. of Utah, Salt Lake City, 1999.

[15] W. Belluomini and C. J. Myers, "Timed state space exploration using POSETs," *IEEE Trans. Computer-Aided Design*, vol. 19, pp. 501–520, May 2000.

[16] C. J. Myers, "Computer-aided synthesis and verification of gate-level timed circuits," Ph.D. dissertation, Stanford Univ., Stanford, CA, 1995.

[17] W. Belluomini and C. J. Myers, "Timed event/level structures," in Collection of papers from TAU'97, pp. 39–44.

[18] P. Vanbekbergen, G. Goossens, and H. de Man, "Specification and analysis of timing constraints in signal transition graphs," in *Proc. Eur. Design Automat. Conf.*, 1992, pp. 72–77.

[19] K. Khordoc and E. Cerny, "Semantics and verification of timing diagrams with linear timing constraints," *ACM Trans. Design Automat. Electron. Syst.*, vol. 3, no. 1, pp. 21–60, Jan. 1998.

[20] A. Valmari, "A stubborn attack on state explosion," in *Int. Conf. Computer-Aided Verification*, June 1990, pp. 176–185.

[21] P. Godefroid, "Using partial orders to improve automatic verification methods," in *Int. Conf. Computer-Aided Verification*, June 1990, pp. 176–185.

[22] K. McMillan, "Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits," in *Lecture Notes in Computer Science*, G. V. Bochman and D. K. Probst, Eds. New York: Springer-Verlag, 1992, vol. 663, pp. 164–177.

[23] J. R. Burch, "Modeling timing assumptions with trace theory," *ICCD*, pp. 29–36, 1989.

[24] D. L. Dill, "Timing assumptions and verification of finite-state concurrent systems," in *Proc. Workshop Automat. Verification Methods Finite-State Syst.*, 1989, pp. 197–212.
[25] B. Berthomieu and M. Diaz, "Modeling and verification of time dependent systems using time petri nets," *IEEE Trans. Software Eng.*, vol. 17, pp. 259–273, Mar. 1991.
[26] H. R. Lewis, "Finite-state analysis of asynchronous circuits with bounded temporal uncertainty," Harvard Univ., Cambridge, MA, Tech. Rep., July 1989.
[27] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi, "Partial order reductions for timed systems," in *Int. Conf. Concurrency Theory*, Sept. 1998, pp. 485–500.
[28] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng, "Automatic synthesis of gate-level timed circuits with choice," in *16th Conference on Advanced Research in VLSI*. Los Alamitos, CA: IEEE Computer Soc. Press, 1995, pp. 42–58.
[29] T. G. Rokicki, "Representing and modeling circuits," Ph.D. dissertation, Stanford Univ., Stanford, CA, 1993.
[30] T. G. Rokicki and C. J. Myers, "Automatic verification of timed circuits," in *Int. Conf. on Computer-Aided Verification*, New York: Springer-Verlag, 1994, pp. 468–480.
[31] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng, "Poset timing and its application to the synthesis and verification of gate-level timed circuit," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 769–786, June 1999.
[32] W. Belluomini and C. J. Myers, "Verification of timed systems using POSETs," in *Int. Conf. on Computer Aided Verification*, New York: Springer-Verlag, 1998, pp. 403–415.
[33] M. R. Greenstreet, "STARI: A technique for high-bandwidth communication," Ph.D. dissertation, Princeton University, Princeton, NJ, 1993.
[34] ——, "Stari: Skew tolerant communication," unpublished.
[35] S. Tasiran and R. K. Brayton, "Stari: A case study in compositional and hierarchical timing verification," in *Proc. Int. Conf. Computer-Aided Verification*, 1997, pp. 191–201.
[36] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello, "Practical applications of an efficient time separation of events algorithm," in *ICCAD*, 1993, pp. 146–151.
[37] R. Alur and R. P. Kurshan, "Timing analysis in cospan," in *Hybrid Systems III*. New York: Springer-Verlag, 1996, pp. 220–231.
[38] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A model-checking tool for real-time systems," in *Proc. Int. Conf. Computer-Aided Verification*, 1998, pp. 546–550.
[39] T. Yoneda and H. Ryu, "Timed trace theoretic verification using partial order reduction," in *Proc. Int. Symp. Advanced Res. Asynchronous Circuits Syst.*, Apr. 1999, pp. 108–121.
[40] C. E. Molnar, I. W. Jones, B. Coates, and J. Lexau, "A FIFO ring oscillator performance experiment," in *Proceedings International Symposium on Advanced Research in Asynchronous Circuits and Systems*. Los Alamitos, CA: IEEE Comput. Soc. Press, Apr. 1997, pp. 279–289.
[41] K. Nowka, T. Galambos, and S. Dhong, "Circuit design techniques for a gigahertz integer microprocessor," in *Int. Conf. Comput. Design*, Oct. 1998, pp. 11–16.
[42] T. I. Chappell, B. A. Chappell, S. E. Schuster, J. W. Allan, S. P. Klepner, R. V. Joshi, and R. L. Franch, "A 2-ns cycle, 3.8-ns access 512-kb CMOS ECL SRAM with a fully pipelined architecture," *IEEE J. Solid-State Circuits*, vol. 26, no. 11, pp. 1577–1585, Nov. 1991.
[43] S. Posluszny *et al.*, "Design methodology for a 1.0 GHz microprocessor," in *Int. Conf. Computer Design*, 1998, pp. 17–23.
[44] J. Silberman *et al.*, "A 1.0 GHz single issue 64-bit powerPC integer processor," *IEEE J. Solid-State Circuits*, vol. 33, pp. 1600–1608, Nov. 1998.
[45] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence relations," *IEEE Trans. Comput.*, vol. 22, pp. 786–793, Aug. 1973.
[46] Y. Kukimoto and R. K. Brayton, "Delay characterization of combinational modules," in *Int. Workshop Logic Synthesis*, 1998, pp. 30–36.

Wendy Belluomini (M'99) received the B.S. degree in computer science from the California Institute of Technology, Pasadena, in 1994, the M.S. degree from the University of Washington, Seattle, in 1996, and the Ph.D. degree from the University of Utah, Salt Lake City, in 1999.

Since 1999, she has been with the IBM Austin Research Laboratory, Austin, TX. Her current interests include symboliic trajectory evaluation, timing verification, and high-speed circuit design.

Dr. Belluomini received a National Science Foundation (NSF) traineeship in 1996 and a Defense Advanced Research Projects Agency ASSERT fellowship in 1997.

Chris J. Myers (S'91–M'96) received the B.S. degree in electrical engineering and Chinese history from the California Institute of Technology, Pasadena, in 1991, and the M.S.E.E. and Ph.D. degrees from Stanford University, Stanford, CA, in 1993 and 1995, respectively.

He has been an Assistant Professor in the Department of Electrical Engineering at the University of Utah, Salt Lake City, since 1995 where he is also the Director for the Center for Asynchronous Circuit and System Design and the Director for the Computer Engineering Program. His current research interests include innovative architectures for high performance and low power, algorithms for the computer-aided analysis and design of real-time concurrent systems, formal verification, and asynchronous circuit design.

Dr. Myers received a National Science Foundation (NSF) Fellowship in 1991, an NSF CAREER award in 1996, and a Best Paper Award at Async99. He is a Co-Organizer and Technical Program Chair for the Async2001 and ARVLSI2001 conferences.

H. Peter Hofstee (M'96) received the Drs. degree from Rijks Universiteit Groningen, Groningen, The Netherlands, and the M.S. and Ph.D. degrees from the California Institute of Technology, Pasadena.

He has been a Research Staff Member with the IBM Austin Research Laboratory, Austin, TX, since 1996, where he works on high-frequency microprocessors. His research interests include areas where microarchitecture and physical design meet.